LNCS 14576 ARCoSS

Stephanie Weirich (Ed.)

Programming Languages and Systems

33rd European Symposium on Programming, ESOP 2024
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2024
Luxembourg City, Luxembourg, April 6–11, 2024
Proceedings, Part I

1 Part I

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

Springer

OPEN ACCESS

Stephanie Weirich
Editor

Programming Languages and Systems

33rd European Symposium on Programming, ESOP 2024
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2024
Luxembourg City, Luxembourg, April 6–11, 2024
Proceedings, Part I

Springer

Editor
Stephanie Weirich
University of Pennsylvania
Philadelphia, PA, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-57261-6 ISBN 978-3-031-57262-3 (eBook)
https://doi.org/10.1007/978-3-031-57262-3

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

ETAPS Foreword

Welcome to the 27th ETAPS! ETAPS 2024 took place in Luxembourg City, the beautiful capital of Luxembourg.

ETAPS 2024 is the 27th instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference established in 1998, and consists of four conferences: ESOP, FASE, FoSSaCS, and TACAS. Each conference has its own Program Committee (PC) and its own Steering Committee (SC). The conferences cover various aspects of software systems, ranging from theoretical computer science to foundations of programming languages, analysis tools, and formal approaches to software engineering. Organising these conferences in a coherent, highly synchronized conference programme enables researchers to participate in an exciting event, having the possibility to meet many colleagues working in different directions in the field, and to easily attend talks of different conferences. On the weekend before the main conference, numerous satellite workshops took place that attracted many researchers from all over the globe.

ETAPS 2024 received 352 submissions in total, 117 of which were accepted, yielding an overall acceptance rate of 33%. I thank all the authors for their interest in ETAPS, all the reviewers for their reviewing efforts, the PC members for their contributions, and in particular the PC (co-)chairs for their hard work in running this entire intensive process. Last but not least, my congratulations to all authors of the accepted papers!

ETAPS 2024 featured the unifying invited speakers Sandrine Blazy (University of Rennes, France) and Lars Birkedal (Aarhus University, Denmark), and the invited speakers Ruzica Piskac (Yale University, USA) for TACAS and Jérôme Leroux (Laboratoire Bordelais de Recherche en Informatique, France) for FoSSaCS. Invited tutorials were provided by Tamar Sharon (Radboud University, the Netherlands) on computer ethics and David Monniaux (Verimag, France) on abstract interpretation.

As part of the programme we had the first ETAPS industry day. The goal of this day was to bring industrial practitioners into the heart of the research community and to catalyze the interaction between industry and academia. The day was organized by Nikolai Kosmatov (Thales Research and Technology, France) and Andrzej Wąsowski (IT University of Copenhagen, Denmark).

ETAPS 2024 was organized by the SnT - Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg. The University of Luxembourg was founded in 2003. The university is one of the best and most international young universities with 6,000 students from 130 countries and 1,500 academics from all over the globe. The local organisation team consisted of Peter Y.A. Ryan (general chair), Peter B. Roenne (organisation chair), Maxime Cordy and Renzo Gaston Degiovanni (workshop chairs), Magali Martin and Isana Nascimento (event manager), Marjan Skrobot (publicity chair), and Afonso Arriaga (local proceedings chair). This team also organised the online edition of ETAPS 2021, and now we are happy that they agreed to also organise a physical edition of ETAPS.

ETAPS 2024 is further supported by the following associations and societies: ETAPS e.V., EATCS (European Association for Theoretical Computer Science), EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology).

I would like to take this opportunity to thank all authors, keynote speakers, attendees, organizers of the satellite workshops, and Springer Nature for their support. ETAPS 2024 was also generously supported by a RESCOM grant from the Luxembourg National Research Foundation (project 18015543). I hope you all enjoyed ETAPS 2024.

Finally, a big thanks to both Peters, Magali and Isana and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

April 2024

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

These proceedings volumes contain papers that were presented at the 33rd European Symposium on Programming (ESOP 2024), held during April 6–11 in Luxembourg City, Luxembourg, along with associated artifact reports. ESOP is part of the European Joint Conferences on Theory and Practice of Software (ETAPS) and promotes the specification, design, analysis and implementation of programming languages and systems.

In total, these two volumes include 25 research papers, one “fresh perspective” and four “artifact reports”. The latter two paper categories are new to ESOP. In addition to standard research papers, the ESOP 2024 call-for-papers included the new submission categories: “fresh perspectives” that provide new insights in a particularly elegant way and “experience reports” that describe tools and systems used in practice. Furthermore, authors of accepted papers were allowed to submit short “artifact reports”, to appear together with their research papers, that describe associated software, tools, data sets, or machine checked proofs to substantiate the claims made in their papers.

The papers in this volume were selected from 66 papers submitted in the research paper category and 6 papers submitted in the “fresh perspectives” category. There were no submissions for “experience reports”. While papers in these new categories had strict formatting requirements, ESOP 2024 allowed research papers to be submitted in any format, of any length, under the advisement that the final paper should be formatted to fit this volume. Fourteen submissions took advantage of this flexibility.

Each submitted paper received at least three reviews by the members of the ESOP program committee. The median PC member was assigned eight papers to review over the seven week review period. In some cases, PC members solicited additional reviews to aid in the decision making process. In total, 39 external reviewers added their insight to the paper selection process. ESOP employed full double-blind review and author identities were only revealed to reviewers on paper acceptance. Authors were also given a chance to respond to their reviews, before the program was selected through a two week online, asynchronous PC meeting, facilitated by the EasyChair system. The program chair had no conflicts with any submitted paper.

ESOP 2024 also employed an artifact evaluation process. Nineteen of the 26 accepted papers elected to make their artifacts available on the archive sites Zenodo and figshare. The committee awarded the badge “Functional” to five of these and the badges “Functional and reusable” to the remaining fourteen. Four accepted papers in this volume are accompanied by artifact reports. These reports were all accepted following a light review by both the program committee and the ESOP/FASE/FoSSaCS joint artifact evaluation committee.

Indeed, my sincere thanks go to all who worked together to produce this event and its proceedings. Foremost, to the authors, who provided the technical content of the meeting. Also to the program committee, artifact evaluation committee, and external reviewers, who provided their well-reasoned and detailed judgments, sometimes on short notice. Tobias Kappé as the representative for ESOP among the artifact evaluation committee co-chairs, deserves particular thanks. I also would like to thank the ETAPS steering committee and its chair Marieke Huisman, the Proceedings coordinator Barbara König and the local proceedings chair Afonso Delerue Arriaga, and webmaster Jan Kofroň for their assistance in fitting ESOP together with the entire ETAPS meeting. Finally, thanks are due to the members of the ESOP steering committee. In particular, Luis Caires, as chair of the SC, was a constant source of support, encouragement, information and guidance.

April 2024

Stephanie Weirich
ESOP PC Chair

Organization

ESOP Steering Committee

Luis Caires	Universidade Nova de Lisboa, Portugal
Brigitte Pientka	McGill University, Canada
Ilya Sergey	Yale-NUS College and National University of Singapore, Singapore
Thomas Wies	New York University, USA
Nobuko Yoshida	Imperial College London, UK
Victor Vafaiedis	Max Planck Institute for Software and Systems, Germany

Program Chair

Stephanie Weirich	University of Pennsylvania, USA

Program Committee

Ana Bove	Chalmers University of Technology, Sweden
Loris D'Antoni	University of Wisconsin-Madison, USA
Ugo Dal Lago	Università di Bologna and Inria Sophia Antipolis, Italy
Ornela Dardha	University of Glasgow, UK
Mike Dodds	University of York, UK
Sophia Drossopoulou	Imperial College London, UK
Robby Findler	Northwestern University, USA
Amir Goharshady	Hong Kong University of Science and Technology, China
Andrew D. Gordon	Microsoft Research and University of Edinburgh, UK
Alexey Gotsman	IMDEA Software Institute, Spain
Limin Jia	Carnegie Mellon University, USA
Josh Ko	Institute of Information Science, Academia Sinica, Taiwan
András Kovács	Eötvös Loránd University, Hungary
Kazutaka Matsuda	Tohoku University, Japan
Anders Miltner	Simon Fraser University, Canada
Santosh Nagarakatte	Rutgers University, USA
Dominic Orchard	University of Kent, UK
Frank Pfenning	Carnegie Mellon University, USA
Clément Pit-Claudel	EPFL, Switzerland
François Pottier	Inria, France
Matija Pretnar	University of Ljubljana, Slovenia
Azalea Raad	Imperial College London, UK

James Riely	DePaul University, USA
Tom Schrijvers	Katholieke Universiteit Leuven, Belgium
Peter Sewell	University of Cambridge, UK
Takeshi Tsukada	Chiba University, Japan
Benoît Valiron	LMF - CentraleSupelec, Univ. Paris Saclay, France
Dimitrios Vytiniotis	Google DeepMind, UK
Elena Zucca	DIBRIS - University of Genova, Italy

ESOP/FASE/FoSSaCS Joint Artifact Evaluation Committee

AEC Co-chairs

Tobias Kappé	Open Universiteit and ILLC, University of Amsterdam, The Netherlands
Ryosuke Sato	University of Tokyo, Japan
Stefan Winter	LMU Munich, Germany

AEC Members

Arwa Hameed Alsubhi	University of Glasgow, UK
Levente Bajczi	Budapest University of Technology and Economics, Hungary
James Baxter	University of York, UK
Matthew Alan Le Brun	University of Glasgow, UK
Laura Bussi	University of Pisa, Italy
Gustavo Carvalho	Universidade Federal de Pernambuco, Brazil
Chanhee Cho	Carnegie Mellon University, USA
Ryan Doenges	Northeastern University, USA
Zainab Fatmi	University of Oxford, UK
Luke Geeson	University College London, UK
Hans-Dieter Hiep	Leiden University, Belgium
Philipp Joram	Tallinn University of Technology, Estonia
Ulf Kargén	Linköping University, Sweden
Hiroyuki Katsura	University of Tokyo, Japan
Calvin Santiago Lee	Reykjavík University, Iceland
Livia Lestingi	Politecnico di Milano, Italy
Nuno Macedo	University of Porto and INESC TEC, Portugal
Kristóf Marussy	Budapest University of Technology and Economics, Hungary
Ivan Nikitin	University of Glasgow, UK
Hugo Pacheco	University of Porto, Portugal
Lucas Sakizloglou	Brandenburgische Technische Universität Cottbus-Senftenberg, Germany
Michael Schröder	TU Wien, Austria
Michael Schwarz	TU Munich, Germany
Wenjia Ye	University of Hong Kong, China

Additional Reviewers

Thorsten Altenkirch
Carlo Angiuli
Martin Avanzini
Aurèle Barrière
Clément Blaudeau
Timothy Bourke
Marco Carbone
Tej Chajed
John Cyphert
Francesco Dagnino
Hoang-Hai Dang
Jana Dunfield
Peter Dybjer
Oskar Eriksson
Simon Fowler
Jose Fragoso Santos
Lorenzo Gheri
Raymond Hu
Patrik Jansson
Delia Kesner
Jinwoo Kim
Robbert Krebbers
Ivan Lanese
Sam Lindley
Peter Ljunglöf
Kenji Maillard
Daniel Marshall
Stephen Mell
Yasuhiko Minamide
Hugo Moeneclaey
Alexandre Moine
Charlie Murphy
Shaan Nagy
Ulf Norell
Mário Pereira
Alejandro Russo
Bernardo Toninho
Paulo Torrens
Ruifeng Xie

Contents – Part I

Contents – Part II

Effects and Modal Types

Scoped Effects as Parameterized Algebraic Theories

Sam Lindley[1], Cristina Matache[1](✉), Sean Moss[2], Sam Staton[3], Nicolas Wu[4], and Zhixuan Yang[4]

[1] University of Edinburgh, Edinburgh, UK
{sam.lindley,cristina.matache}@ed.ac.uk
[2] University of Birmingham, Birmingham, UK
s.k.moss@bham.ac.uk
[3] University of Oxford, Oxford, UK
sam.staton@cs.ox.ac.uk
[4] Imperial College London, London, UK
{n.wu,s.yang20}@imperial.ac.uk

Abstract. Notions of computation can be modelled by monads. *Algebraic effects* offer a characterization of monads in terms of algebraic operations and equational axioms, where operations are basic programming features, such as reading or updating the state, and axioms specify observably equivalent expressions. However, many useful programming features depend on additional mechanisms such as delimited scopes or dynamically allocated resources. Such mechanisms can be supported via extensions to algebraic effects including *scoped effects* and *parameterized algebraic theories*. We present a fresh perspective on scoped effects by translation into a variation of parameterized algebraic theories. The translation enables a new approach to equational reasoning for scoped effects and gives rise to an alternative characterization of monads in terms of generators and equations involving both scoped and algebraic operations. We demonstrate the power of our fresh perspective by way of equational characterizations of several known models of scoped effects.

Keywords: algebraic effects · scoped effects · monads · category theory · algebraic theories.

1 Introduction

The central idea of *algebraic effects* [29] is that impure computation can be built and reasoned about equationally, using an algebraic theory. *Effect handlers* [28] are a way of implementing algebraic effects and provide a method for modularly programming with different effects. More formally, an effect handler gives a model for an algebraic theory. In this paper we develop equational reasoning for a notion arising from an extension of handlers, called *scoped effects*, using the framework of *parameterized algebraic theories*.

The central idea of *scoped effects* (Sec. 2.2) is that certain parts of an impure computation should be dealt with one way, and other parts another way, inspired by scopes in exception handling. Compared to algebraic effects, the crucial difference is that the scope on which a scoped effect acts is delimited. This difference leads to a complex relationship with monadic sequencing ($\gg=$). The theory and practice of scoped effects [41,23,42,5,40,43] has primarily been studied by extending effect handlers to deal with not just algebraic operations, but also more complex scoped operations. They form the basis of the `fused-effects` and `polysemy` libraries for Haskell. Aside from exception handling, other applications include back-tracking in parsing [41] and timing analysis in telemetry [39].

Parameterized algebraic theories (Sec. 2.3) extend plain algebraic theories with variable binding operations for an abstract type of parameters. They have been used to study various resources including logic variables in logic programming [35], channels in the π-calculus [36], code pointers [7], qubits in quantum programming [38], and urns in probabilistic programming [34].

Contributions. We propose an equational perspective for scoped effects where *scopes are resources*, by analogy with other resources like file handles. We develop this perspective using the framework of *parameterized algebraic theories*, which provides an algebraic account of effects with resources and instances. We realize scoped effects by encoding the scopes as resources with open/close operations, analogous to opening/closing files. This fresh perspective provides:

- the first syntactic sound and complete equational reasoning system for scoped effects, based on the equational reasoning for parameterized algebraic theories (Prop. 2, Prop. 3);
- a canonical notion of semantic model for scoped effects supporting three key examples from the literature: nondeterminism with semi-determinism (Thm. 2), catching exceptions (Thm. 3), and local state (Thm. 4); and
- a reconstruction of the previous categorical analysis of scoped effects via the categorical analysis of parameterized algebraic theories: the constructors $(\triangleleft, \triangleright)$ are shown to be not ad hoc, but rather the crucial mechanism for arities/coarities in parameterized algebraic theories (Thm. 1).

Example: nondeterminism with semi-determinism. We now briefly illustrate the intuition underlying the connection between scoped effects and parameterized algebraic theories through an example. (See Examples 1 and 4 for further details.) Let us begin with two algebraic operations: $\mathsf{or}(x, y)$, which nondeterministically chooses between continuing[5] as computation x or as computation y, and fail, which fails immediately. We add semi-determinism in the form of a *scoped* operation $\mathsf{once}(x)$, which chooses the first branch of the computation x that does not fail. Importantly, the scope that once

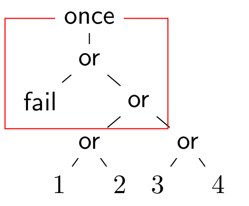


Fig. 1. Illustrating (1)

[5] This continuation-passing style is natural for algebraic effects, but when programming one often uses equivalent direct-style *generic effects* [25] such as $\underline{\mathsf{or}} : \mathsf{unit} \to \mathsf{bool}$, where $\mathsf{or}(x, y)$ can be recovered by pattern matching on the result of $\underline{\mathsf{or}}$.

acts on is delimited. The left program below returns 1; the right one returns 1 or 2, as the second or is outside the scope of once.

$$\mathsf{once}(\mathsf{or}(\mathsf{or}(1,2),\mathsf{or}(3,4))) \qquad \mathsf{once}(\mathsf{or}(1,3)) \gg\!= \lambda x.\, \mathsf{or}(x, x+1)$$

Now consider a slightly more involved example, which also returns 1 or 2:

$$\mathsf{once}(\mathsf{or}(\mathsf{fail},\ \mathsf{or}(1,3))) \gg\!= \lambda x.\, \mathsf{or}(x, x+1) \tag{1}$$

depicted as a tree in Fig. 1 where the red box delimits the scope of once. We give an encoding of term (1) in a parameterized algebraic theory as follows:

$$\mathsf{once}(a.\mathsf{or}(\mathsf{fail}, \mathsf{or}(\mathsf{close}(a, \mathsf{or}(1,2)), \mathsf{close}(a, \mathsf{or}(3,4))))) \tag{2}$$

where a is the name of the scope opened by once and closed by the special close operation. By equational reasoning for scoped effects (§3) and the equations for nondeterminism (Fig. 2), we can prove that the term (2) is equivalent to $\mathsf{or}(1,2)$.

2 Background

2.1 Algebraic effects

Moggi [20,21] shows that many non-pure features of programming languages, typically referred to as *computational effects*, can be modelled uniformly as *monads*, but the question is — *how do we construct a monad for an effect*, or putting it differently, *where do the monads modelling effects come from*? A classical result in category theory is that finitary monads over the category of sets are equivalent to *algebraic theories* [16,15]: an algebraic theory gives rise to a finitary monad by the free-algebra construction, and conversely every finitary monad is presented by a certain algebraic theory. Motivated by this correspondence, Plotkin and Power [26] show that many monads that are used for modelling computational effects can be presented by algebraic theories of some basic effectful operations and some computationally natural equations. This observation led them to the following influential perspective on computational effects [26], which is nowadays commonly referred to as *algebraic effects*:

Perspective 1 ([26]). An effect is realized by an algebraic theory of its basic operations, so it *determines* a monad but is not identified with the monad.

We review the framework in a simple form here; see [27,2] for more discussion.

Definition 1. *A (first-order finitary)* algebraic signature $\Sigma = \langle |\Sigma|, ar\rangle$ *consists of a set* $|\Sigma|$*, whose elements are referred to as* operations*, together with a mapping* $ar : |\Sigma| \to \mathbb{N}$*, associating an* arity *to each operation.*

Given a signature $\Sigma = \langle |\Sigma|, ar\rangle$, we will write $O : n$ for an operation $O \in |\Sigma|$ with $ar(O) = n$. The *terms* $\mathsf{Tm}_\Sigma(\Gamma)$ in a context Γ, which is a finite list of variables, are inductively generated by the following rules:

$$\frac{}{\Gamma, x, \Gamma' \vdash x} \qquad \frac{(O : n) \qquad \Gamma \vdash t_i \text{ for } i = 1 \ldots n}{\Gamma \vdash O(t_1, \ldots, t_n)}$$

As usual we will consider terms up to renaming of variables. Thus a context $\Gamma = (x_1, \ldots, x_n)$ can be identified with the natural number n, and Tm_Σ can be thought of as a function $\mathbb{N} \to \mathbf{Set}$.

Example 1. The signature of *explicit nondeterminism* has two operations:

$$\mathsf{or} : 2 \qquad\qquad \mathsf{fail} : 0.$$

Some small examples of terms of this signature are

$$\vdash \mathsf{fail} \qquad x, y, z \vdash \mathsf{or}(x, \mathsf{or}(y, z)) \qquad x, y, z \vdash \mathsf{or}(\mathsf{or}(x, y), \mathsf{fail})$$

Example 2. The signature of *mutable state* of a single bit has operations:

$$\mathsf{put}^0 : 1 \qquad \mathsf{put}^1 : 1 \qquad \mathsf{get} : 2.$$

The informal intuition for a term $\Gamma \vdash \mathsf{put}^i(t)$ is a program that writes the bit $i \in \{0, 1\}$ to the mutable state and then continues as another program t, and a term $\Gamma \vdash \mathsf{get}(t_0, t_1)$ is a program that reads the state, and continues as t_i if the state is i. For example, the term $x, y \vdash \mathsf{put}^0(\mathsf{get}(x, y))$ first writes 0 to the state, then reads 0 from the state, so always continues as x. For simplicity we consider a single bit, but multiple fixed locations and other storage are possible [26].

Definition 2. *A (first-order finitary)* algebraic theory $T = \langle \Sigma, E\rangle$ *is a signature Σ (Def. 1) and a set E of* equations *of the signature Σ, where an equation is a pair of terms $\Gamma \vdash L$ and $\Gamma \vdash R$ under some context Γ. We will usually write an equation as $\Gamma \vdash L = R$.*

Example 3. The theory of *exception throwing* has a signature containing a single operation $\mathsf{throw} : 0$ and no equations. The intuition for throw is that it throws an exception and the control flow never comes back, so it is a nullary operation.

Example 4. The theory of *explicit nondeterminism* has the signature in Example 1 and the following equations saying that fail and or form a monoid:

$$x \vdash \mathsf{or}(\mathsf{fail}, x) = x \quad x \vdash \mathsf{or}(x, \mathsf{fail}) = x \quad x, y, z \vdash \mathsf{or}(x, \mathsf{or}(y, z)) = \mathsf{or}(\mathsf{or}(x, y), z)$$

Example 5. The theory of *mutable state* has the signature in Example 2 and the following equations for all $i, i' \in \{0, 1\}$:

$$x_0, x_1 \vdash \mathsf{put}^i(\mathsf{get}(x_0, x_1)) = \mathsf{put}^i(x_i) \qquad x \vdash \mathsf{put}^i(\mathsf{put}^{i'}(x)) = \mathsf{put}^{i'}(x)$$

$$x \vdash \mathsf{get}(\mathsf{put}^0(x), \mathsf{put}^1(x)) = x$$

Every algebraic theory gives rise to a monad by the *free-algebra construction*, which we will discuss in a more general setting in Section 3. The three examples above respectively give rise to the monads $(1 + -)$, List, $(- \times \mathbb{2})^{\mathbb{2}}$ on the category of sets that are used to give semantics to the respective computational effects in programming languages [20,21]. In this way, the monad for a computational effect is constructed in a very intuitive manner, and this approach is highly composable: one can take the disjoint union of two algebraic theories to combine two effects, and possibly add more equations to characterise the interaction between the two theories [12]. By contrast, monads are not composable in general.

The kind of plain algebraic theory encapsulated by Def. 2 above is not, however, sufficiently expressive enough for some programming language applications. In this paper we focus on two problems with plain algebraic theories:

1. Firstly, monadic bind for the monad generated by an algebraic theory is essentially defined using *simultaneous substitution of terms*: given a term $t \in \mathsf{Tm}(\Gamma)$ in a context Γ and a mapping $\sigma : \Gamma \to \mathsf{Tm}(\Gamma')$ from variables in Γ to terms in some context Γ', the simultaneous substitution of σ in t is $t[\sigma]$ where
$$x[\sigma] = \sigma(x) \qquad\qquad O(t_1, \ldots, t_n)[\sigma] = O(t_1[\sigma], \ldots, t_n[\sigma]).$$
On the other hand, bind for a monad is used for interpreting *sequential composition* of computations. Therefore, the second clause above implies that every algebraic effect operation *must* commute with sequential composition. However, in practice not every effectful operation enjoys this property.
2. Secondly, it is common to have *multiple instances* of a computational effect that can be dynamically created. For example, it is typical in practice to have an effectful operation $\mathsf{openFile}$ that creates a 'file descriptor' for a file at a given path, and for each file descriptor there is a pair of read and write operations that are independent of those for other files.

These two restrictions have been studied separately, and different extensions to algebraic theories generalising Def. 2 have been proposed for each: *scoped algebraic effects* for the first problem above and *parameterized algebraic effects* for the second. At first glance, the two problems seem unrelated, but the fresh perspective of this paper is that scoped effects can be fruitfully understood as a *non-commutative linear* variant of parameterized effects.

2.2 Scoped effects

Recall that our first problem with plain algebraic theories is that operations must commute with sequential composition. Therefore an operation $O(a_1, \ldots, a_n)$ is 'atomic' in the sense that it may not delimit a fresh *scope*. Alas, in practice it is not uncommon to have operations that do delimit scopes. An example is *exception catching*: $\mathsf{catch}(p, h)$ is a binary operation on computations that first tries the program p and if p throws an exception then h is run. The catch operation does not commute with sequential composition as $\mathsf{catch}(p, h) \gg\!= f$ behaves differently from $\mathsf{catch}(p \gg\!= f, h \gg\!= f)$. The former catches only the exceptions in p whereas the latter catches exceptions both in p and in f. Further examples include operations such as opening a file in a scope, running a program concurrently in a scope, and looping a program in a scope.

Operations delimiting scopes are treated as *handlers* (i.e. models) of algebraic operations by Plotkin and Pretnar [28], instead of operations in their own right. The following alternative perspective was first advocated by Wu et al. [41].

Perspective 2 ([41]). Scoped operations are *operations that do not commute with substitution*, since sequential composition in monads generated from algebraic theories corresponds to *substitution.* Such operations arise in contexts other than computational effects as well, for example, the later modality in *guarded dependent type theory* (GDTT) [4].

Extensions of algebraic effects to accommodate scoped operations were first studied by Wu et al. [41] in Haskell, where the authors proposed two approaches:

1. The *bracketing approach* uses a pair of *algebraic* operations begin_s and end_s to encode a scoped operation s. For example, the program $s(\mathsf{put}^0); \mathsf{put}^1(x)$, where put^0 is wrapped in the scope of s, is encoded formally as

$$\mathsf{begin}_s(\mathsf{put}^0(\mathsf{end}_s(\mathsf{put}^1(x)))).$$

2. The *higher-order abstract syntax (HOAS) approach* directly constructs a monad for programs with algebraic and scoped operations. In Haskell, their monad for programs with algebraic operations parameterized by a signature functor `asig` and scoped operations parameterized by a functor `ssig` is

```
data Prog a where
  Ret :: a -> Prog a
  Alg :: asig (Prog a) -> Prog a
  Scp :: forall x. ssig (Prog x) -> (x -> Prog a) -> Prog a
```

   where `Scp p f` represents a scoped operation acting on a program `p` followed by a program `f` after the scope (cf *delayed substitution* in GDTT [4]).

The HOAS approach was regarded the more principled one since in the first approach ill bracketed pairs of begin_s and end_s are possible, such as

$$\mathsf{end}_s(\mathsf{put}^0(\mathsf{begin}_s(\mathsf{begin}_s(\mathsf{put}^1(x))))).$$

In subsequent work, both of these two approaches received further development [23,43,40,42] and operational semantics for scoped effects has also been developed [5]. Of particular relevance to the current paper is the work of Piróg et al. [23], which we briefly review in the rest of this section.

Piróg et al. [23] fix the ill-bracketing problem in the bracketing approach by considering the category $\mathbf{Set}^{\mathbb{N}}$ whose objects are sequences $X = (X(0), X(1), \ldots)$ of sets and morphisms are just sequences of functions. Given $X \in \mathbf{Set}^{\mathbb{N}}$, the idea is that $X(n)$ represents a set of terms at *bracketing level* n for every $n \in \mathbb{N}$.

On this category, there are two functors $(\triangleright), (\triangleleft) : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbb{N}}$, pronounced 'later' and 'earlier', that shift the bracketing levels:

$$(\triangleright X)(0) = \emptyset, \qquad (\triangleright X)(n+1) = X(n), \qquad (\triangleleft X)(n) = X(n+1). \tag{3}$$

These two functors are closely related to bracketing: a morphism $b : \triangleleft X \to X$ for a functor X opens a scope, turning a term t at level $n+1$ to the term $\mathsf{begin}(t)$ at level n. Conversely, a morphism $e : \triangleright X \to X$ closes a scope, turning a term t outside the scope, so at level $n-1$, to the term $\mathsf{end}(t)$ at level n.

Given two signatures Σ and Σ' as in Def. 1 for algebraic and scoped operations respectively, let $\bar{\Sigma}, \bar{\Sigma}' : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbb{N}}$ be the functors given by

$$(\bar{\Sigma}X)(n) = \coprod_{o \in |\Sigma|} X(n)^{ar(o)} \quad \text{and} \quad (\bar{\Sigma}'X)(n) = \coprod_{s \in |\Sigma'|} X(n)^{ar(s)}.$$

Moreover, for every $A \in \mathbf{Set}$, let $\upharpoonleft A \in \mathbf{Set}^{\mathbb{N}}$ be given by

$$(\upharpoonleft A)(0) = A \qquad\qquad (\upharpoonleft A)(n+1) = 0,$$

and conversely for every $X \in \mathbf{Set}^{\mathbb{N}}$, let $\downharpoonleft X \in \mathbf{Set}$ be given by $\downharpoonleft X = X(0)$.

Proposition 1 (Piróg et al. [23]). *The following functor can be extended to a monad that is isomorphic to the monad* Prog *in the HOAS approach above:*

$$\downharpoonleft \circ \left(\bar{\Sigma} + \left(\bar{\Sigma}' \circ \triangleleft\right) + \triangleright\right)^* \circ \upharpoonleft : \mathbf{Set} \to \mathbf{Set}$$

where $(-)^*$ *is the free monad over an endofunctor.*

The monad from Prop. 1 is a way of specifying the syntax of programs with algebraic and scoped operations, without taking into account equations. In [23], a *model* of a scoped effect is an *algebra* for the monad $\left(\bar{\Sigma} + \left(\bar{\Sigma}' \circ \triangleleft\right) + \triangleright\right)^*$. In Thms. 2–4, we show that three examples of models from [23] are *free algebras* on $\upharpoonleft A \in \mathbf{Set}^{\mathbb{N}}$ for an appropriate set of equations for each example.

2.3 Parameterized algebraic theories

Recall that our second problem with plain algebraic theories is that they do not support the dynamic creation of multiple instances of computational effects. This problem, sometimes known as the *local computational effects* problem, was first systematically studied by Power [32] in a purely categorical setting. A syntactic framework extending that of algebraic theories, called *parameterized algebraic theories*, was introduced by Staton [35,36] and is used to give an axiomatic account of local computational effects such as restriction [24], local state [26], and the π-calculus [19,33].

Operations in a parameterized theory are more general than those in an algebraic theory because they may *use* and *create* values in an *abstract* type of parameters. The parameter type has different intended meanings for different examples of parameterized theories, typically as some kind of resource such as memory locations or communication channels. In this paper, we propose to interpret parameters as *names of scopes*.

Perspective 3. Scoped operations can be understood as operations allocating and consuming instances of a resource: the names of scopes.

In the case of local state, the operations of Example 2 become $\mathsf{get}(a, x_0, x_1)$ and $\mathsf{put}^i(a, x)$, now taking a parameter a which is the location being read or written to. In a sense, each memory location a represents an *instance* of the state effect, with its own get and put operations. We also have a term $\mathsf{new}^i(a.x(a))$ which allocates a fresh location named a storing an initial value i, then continues as x; the computation x might mention location a. The following is a possible equation, which says that reading immediately after allocating is redundant:

$$\mathsf{new}^i(a.\mathsf{get}(a, x_0(a), x_1(a))) = \mathsf{new}^i(a.x_i(a)).$$

For the full axiomatization of local state see [36, §V.E]. A closed term can only mention locations introduced by new^i, meaning that type of locations is *abstract*.

To model scoped operations, we think of them as allocating a new scope. For example, the scoped operation once, which chooses the first non-failing branch of a nondeterministic computation, is written as $\mathsf{once}(a.x(a))$. It creates a new scope a and proceeds as x. As in §1, there is an explicit operation $\mathsf{close}(a, x)$ for closing the scope a and continuing as x.

Well-formed programs close scopes precisely once and in the reverse order to their allocation. Thus in §3 we will discuss a *non-commutative linear* variation of parameterized algebraic theories needed to model scoped effects. With our framework we then give axiomatizations for examples from the scoped effects literature (Thms. 2–4).

Our parameters are linear in the same sense as variables in linear logic and linear lambda calculi e.g. [11,3], but with an additional non-commutativity restriction. Non-commutative linear systems are also known as ordered linear systems e.g. [30,22]. A commutative linear version of parameterized algebraic theories was considered in [38] to give an algebraic theory of quantum computation; in this case, parameters stand for qubits.

Remark 1. Parameterized algebraic theories characterize a certain class of enriched monads [35], extending the correspondence between algebraic theories and monads on the category of sets, and the idea of Plotkin and Power [26] that computational effects give rise to monads (see §2.1). Thus, the syntactic framework of parameterized theories has a canonical semantic status. We can use the monad arising from a parameterized theory to give semantics to a programming language containing the effects in question.

The framework of parameterized algebraic theories is related to graded theories [13], which also use presheaf-enrichment; second-order algebra [8,9,10], which also use variable binding; and graphical methods [17], which also connect to presheaf categories.

3 Parameterized theories of scoped effects

In order to describe scoped effects we use a substructural version of parameterized algebraic theories [35]. A theory consists of a signature (Def. 3) and

$$x : 0, y : 0, z : 0 \mid - \vdash \mathsf{or}(\mathsf{or}(x, y), z) = \mathsf{or}(x, \mathsf{or}(y, z)) \qquad (4)$$

$$x : 0 \mid - \vdash \mathsf{or}(x, \mathsf{fail}) = x \qquad x : 0 \mid - \vdash \mathsf{or}(\mathsf{fail}, x) = x \qquad (5)$$

$$- \mid - \vdash \mathsf{once}(a.\mathsf{fail}) = \mathsf{fail} \qquad x{:}1 \mid - \vdash \mathsf{once}(a.\mathsf{or}(x(a),\ x(a))) = \mathsf{once}(a.x(a)) \quad (6)$$

$$x{:}0 \mid - \vdash \mathsf{once}(a.\mathsf{close}(a, x)) = x \qquad x{:}0, y{:}1 \mid - \vdash \mathsf{once}\big(a.\mathsf{or}(\mathsf{close}(a, x),\ y(a))\big) = x \quad (7)$$

Fig. 2. The parameterized theory of explicit nondeterminism (4–5) and once (6–7). Terms-in-context are defined further down.

equations (Def. 4) between terms formed from the signature. Terms contain two kinds of variables: computation variables (x, y, ...), which each expect a certain number of parameters, and parameter variables (a, b, ...). In the case of scoped effects, a parameter represents the name of a scope.

Definition 3. *A* (parameterized) signature $\Sigma = \langle |\Sigma|, ar \rangle$ *consists of a set of operations* $|\Sigma|$ *and for each operation* $\mathrm{O} \in |\Sigma|$ *a* parameterized arity $ar(\mathrm{O}) = (p \mid m_1 \ldots m_k)$ *consisting of a natural number* p *and a list of natural numbers* $m_1, \ldots, m_k$*. This means that the operation* O *takes in* p *parameters and* k *continuations, and it binds* m_i *parameters in the* i*-th continuation.*

Remark 2. Given signatures for algebraic and scoped operations, as in Def. 1 and §2.2, we can translate them to a parameterized signature as follows:

– for each algebraic operation $(\mathsf{op} : k)$ of arity $k \in \mathbb{N}$, there is a parameterized operation with arity $(0 \mid 0 \ldots 0)$, where the list $0 \ldots 0$ has length k;
– for each scoped operation $(\mathsf{sc} : k)$ of arity $k \in \mathbb{N}$, there is a parameterized operation $\mathsf{sc} : (0 \mid 1 \ldots 1)$, where the list $1 \ldots 1$ has length k;
– there is an operation close : $(1 \mid 0)$, which closes the most recent scope, and which all the different scoped operations share.

Example 6. The algebraic theory of explicit nondeterminism in Example 1 can be extended with a *semi-determinism* operator once:

$$\mathsf{or} : (0 \mid 0, 0) \qquad \mathsf{once} : (0 \mid 1) \qquad \mathsf{fail} : (0 \mid -) \qquad \mathsf{close} : (1 \mid 0)$$

The continuation of once opens a new scope, by binding a parameter. Inside this scope, only the first successful branch of or is kept. The term formation rules below allow the most recently opened scope to be closed using the close operation by consuming the most recently bound parameter; close has one continuation which does not depend on any parameters. See Fig. 2 for equations.

For a given signature, we define the terms-in-context of *algebra with non-commutative linear parameters.* A context Γ of *computation variables* is a finite list $x_1 : p_1, \ldots, x_n : p_n$, where each variable x_i is annotated with the number p_i of parameters it consumes. A context Δ of *parameter variables* is a finite list

$a_1, \ldots, a_m$. Terms $\Gamma \mid \Delta \vdash t$ are inductively generated by the following two rules.

$$\frac{}{\Gamma, x : p, \Gamma' \mid a_1 \ldots a_p \vdash x(a_1 \ldots a_p)}$$

$$\frac{\Gamma \mid \Delta, b_1 \ldots b_{m_1} \vdash t_1 \qquad \ldots \qquad \Gamma \mid \Delta, b_1 \ldots b_{m_k} \vdash t_k \qquad \mathrm{O} : (p \mid m_1 \ldots m_k)}{\Gamma \mid \Delta, a_1 \ldots a_p \vdash \mathrm{O}(a_1 \ldots a_p,\ b_1 \ldots b_{m_1}.t_1\ \ldots\ b_1 \ldots b_{m_k}.t_k)}$$

In the conclusion of the last rule, the parameters $a_1 \ldots a_p$ are consumed by the operation O. The parameters $b_1 \ldots b_{m_i}$ are bound in t_i. As usual, we treat all terms up to renaming of variables.

The context Γ of computation variables admits the usual structural rules: weakening, contraction, and exchange; the context Δ of parameters does not. All parameters in Δ must be used exactly once, in the reverse of the order in which they appear. Intuitively, a parameter in Δ is the name of an open scope, so the restrictions on Δ mean that scopes must be closed in the opposite order that they were opened, that is, scopes are well-bracketed. The arguments $t_1, \ldots, t_k$ of an operation O are continuations, each corresponding to a different branch of the computation, hence they share the parameter context Δ.

Compared to the *algebra with linear parameters* of [38], used for describing quantum computation, our syntactic framework has the additional constraint that Δ cannot be reordered. Given these constraints, the context Δ is in fact a stack, so inside a term it is unnecessary to refer to the variables in Δ by name. We have chosen to do so anyway in order to make more clear the connection to non-linear parameterized theories [35,36].

The syntax admits the following simultaneous substitution rule:

$$\frac{\begin{array}{c}(x_1 : m_1 \ldots x_l : m_l) \mid \Delta \vdash t \\ \Gamma' \mid \Delta', a_1 \ldots a_{m_1} \vdash t_1 \qquad \ldots \qquad \Gamma' \mid \Delta', a_1 \ldots a_{m_l} \vdash t_l\end{array}}{\Gamma' \mid \Delta', \Delta \vdash t\big[(\Delta', a_1 \ldots a_{m_1} \vdash t_1)/x_1\ \ldots\ (\Delta', a_1 \ldots a_{m_l} \vdash t_l)/x_l\big]} \tag{8}$$

In the conclusion, the notation $(\Delta', a_1 \ldots a_{m_i} \vdash t_i)/x_i$ emphasizes that the parameters $(a_1 \ldots a_{m_i})$ in t_i are replaced by the corresponding parameters that x_i consumes in t, either bound parameters or free parameters from Δ. To ensure that the term in the conclusion is well-formed, we must substitute a term that depends on Δ' for *all* the computation variables in the context of t.

An important special case of the substitution rule is where we add a number of extra parameter variables to the beginning of the parameter context, increasing the sort of each computation variable by the same number. The following example instance of (8), where $ar(\mathrm{O}) = (1 \mid 1)$, illustrates such a 'weakening' by adding two extra parameter variables a'_1, a'_2 and replacing $x : 2$ by $x' : 4$.

$$\frac{x : 2 \mid a_1, a_2 \vdash \mathrm{O}(a_2, b.x(a_1, b)) \qquad x' : 4 \mid a'_1, a'_2, b_1, b_2 \vdash x'(a'_1, a'_2, b_1, b_2)}{x' : 4 \mid a'_1, a'_2, a_1, a_2 \vdash \mathrm{O}(a_2, b.x'(a'_1, a'_2, a_1, b))}$$

Definition 4. *An* algebraic theory $\mathcal{T} = \langle \Sigma, E \rangle$ with non-commutative linear parameters *is a parameterized signature Σ together with a set E of* equations*. An equation is a pair of terms in the same context $(\Gamma \mid \Delta)$ for some Γ and Δ.*

We will omit the qualifier "with non-commutative linear parameters" where convenient and refer to "parameterized theories" or just "theories". Given a theory $\mathcal{T}$, we form a system of equivalence relations $=_{\mathcal{T},(\Gamma|\Delta)}$ on terms in each context $(\Gamma \mid \Delta)$ by closing substitution instances of the axioms under reflexivity, symmetry, transitivity, and congruence.

Example 7. As we mentioned earlier, *exception catching* is not an ordinary algebraic operation. As parameterized operations, the signature for throwing and catching exceptions is the following:

$$\mathsf{throw} : (0 \mid -) \qquad \mathsf{catch} : (0 \mid 1, 1) \qquad \mathsf{close} : (1 \mid 0)$$

The throw operation uses no parameters and takes no continuations. The catch operation uses no parameters and takes two continuations which each open a new scope, by binding a fresh parameter. Exceptions are caught in the first continuation, and are handled using the second continuation.

The close operation uses one parameter and takes one continuation binding no parameters. The term $\mathsf{close}(a, x)$ closes the scope named by a and continues as x. For example, in $\mathsf{catch}(a.\mathsf{close}(a, x), b.y(b))$, exceptions in x will not be caught, because the scope of the catch has already been closed. The equations are:

$$y{:}0 \mid - \vdash \mathsf{catch}(a.\mathsf{throw},\ b.\mathsf{close}(b, y)) = y \tag{9}$$

$$- \mid - \vdash \mathsf{catch}(a.\mathsf{throw},\ b.\mathsf{throw}) = \mathsf{throw} \tag{10}$$

$$x{:}0, y{:}1 \mid - \vdash \mathsf{catch}\big(a.\mathsf{close}(a, x),\ b.y(b)\big) = x \tag{11}$$

Remark 3. The arity of catch from Ex. 7 corresponds to the signature used in [23, Ex. 4.5]. Using the extra flexibility of parameterized algebraic theories, we could instead consider the arity $\mathsf{catch} : (0 \mid 1, 0)$. This seems more natural as there is no need to delimit a scope in the second continuation, which handles the exceptions.

Example 8 (Mutable state with local values). The theory of (boolean) mutable state with one memory location (Ex. 2) can be extended with scoped operations local^0 and local^1 that write respectively 0 and 1 to the state. Inside the scope of local, the value of the state just before the local is not accessible anymore, but when the local is closed the state reverts to this previous value.

$$\mathsf{local}^i : (0 \mid 1) \qquad \mathsf{put}^i : (0 \mid 0) \qquad \mathsf{get} : (0 \mid 0, 0) \qquad \mathsf{close} : (1 \mid 0)$$

The equations for the parameterized theory of state with local comprise the usual equations for state [26,18]:

$$z : 0 \mid - \vdash \mathsf{get}(\mathsf{put}^0(z), \mathsf{put}^1(z)) = z \quad z : 0 \mid - \vdash \mathsf{put}^i(\mathsf{put}^j(z)) = \mathsf{put}^j(z) \tag{12}$$

$$x_0 : 0, x_1 : 0 \mid - \vdash \mathsf{put}^i(\mathsf{get}(x_0, x_1)) = \mathsf{put}^i(x_i) \tag{13}$$

together with equations for local/close, and the interaction with state:

$$x : 0 \mid - \vdash \mathsf{local}^i(a.\mathsf{close}(a, x)) = x \tag{14}$$

$$x_0 : 1, x_1 : 1 \mid - \vdash \mathsf{local}^i(a.\mathsf{get}(x_0(a),\ x_1(a))) = \mathsf{local}^i(a.x_i(a)) \tag{15}$$

$$z : 1 \mid - \vdash \mathsf{local}^i(a.\mathsf{put}^j(z(a))) = \mathsf{local}^j(a.z(a)) \tag{16}$$

$$z : 0 \mid a \vdash \mathsf{put}^i(\mathsf{close}(a, z)) = \mathsf{close}(a, z) \tag{17}$$

This extension of mutable state is different from the one discussed in §2.3, where memory locations can be dynamically created.

4 Models of parameterized theories

4.1 Models in $\mathbf{Set}^{\mathbb{N}}$

Models of first-order algebraic theories [2] consist simply of a set together with specified interpretations of the operations of the signature, validating a (possibly empty) equational specification. The more complex arities and judgement forms of a parameterized theory require a correspondingly more complex notion of model. Rather than simply being a set of abstract computations, a model will now be stratified into a sequence of sets $X = (X(0), X(1), \ldots) \in \mathbf{Set}^{\mathbb{N}}$ where $X(n)$ represents computations *taking n parameters*. In §2.2 we described the use of $\mathbf{Set}^{\mathbb{N}}$ in [23]. We connect the two approaches in Thm. 1 below.

At first glance, a term $x_1 : m_1, \ldots, x_k : m_k \mid a_1, \ldots, a_p \vdash t$ should denote a function $X(m_1) \times \ldots \times X(m_k) \to X(p)$, since a k-tuple of possible continuations that consume different numbers of parameters is mapped to a computation that consumes p parameters. However, the admissible substitution rule (8) shows us that actually such a term must also denote a sequence of functions

$$[\![x_1 : m_1, \ldots, x_k : m_k \mid a_1, \ldots, a_p \vdash t]\!]_{\mathcal{X},n} : X(n{+}m_1) \times \ldots \times X(n{+}m_k) \to X(n{+}p).$$

Definition 5. *Let Σ be a parameterized signature (Def. 3). A Σ*-structure *$\mathcal{X}$ is an $X \in \mathbf{Set}^{\mathbb{N}}$ equipped with, for each $\mathrm{O} : (p \mid m_1 \ldots m_k)$ and $n \in \mathbb{N}$, a function*

$$\mathrm{O}_{\mathcal{X},n} : X(n + m_1) \times \ldots \times X(n + m_k) \to X(n + p).$$

The interpretation of terms is now defined by structural recursion in a standard way, where the interpretation of a computation variable term such as $x_1 : m_1, \ldots, x_k : m_k \mid a_1, \ldots, a_{m_i} \vdash x_i(a_1, \ldots, a_{m_i})$ is given by the sequence of product projections

$$X(n + m_1) \times \ldots \times X(n + m_i) \times \ldots \times X(n + m_k) \to X(n + m_i).$$

Definition 6. *Let $\mathcal{T}$ be a parameterized theory over the signature Σ. A Σ-structure $\mathcal{X}$ is a* model *of $\mathcal{T}$ if for every equation $\Gamma \mid \Delta \vdash s = t$ in $\mathcal{T}$, and every $n \in \mathbb{N}$, we have an equality of functions $[\![\Gamma \mid \Delta \vdash s]\!]_{\mathcal{X},n} = [\![\Gamma \mid \Delta \vdash t]\!]_{\mathcal{X},n}$.*

Proposition 2. *The derivable equality ($=_{\mathcal{T}}$) in a parameterized algebraic theory $\mathcal{T}$ is sound: every $\mathcal{T}$-model satisfies every equation of $=_{\mathcal{T}}$.*

Proof (notes). By induction on the structure of derivations.

Remark 4. A more abstract view on models is based on enriched categories, since parameterized algebraic theories can be understood in terms of enriched Lawvere theories [31,14,35]. This is useful because, by interpreting algebraic theories in different categories, we can combine the algebra structure with other structure, such as topological or order structure for recursion [1, §6], or make connections with syntactic categories [37]. Recall that the category $\mathbf{Set}^{\mathbb{N}}$ has a 'Day convolution' monoidal structure [6]: $(X \otimes Y)(n) = \sum_{m_1+m_2=n} X(m_1) \times Y(m_2)$. With this structure, we can interpret a parameterized algebraic theory $\mathcal{T}$ in any $\mathbf{Set}^{\mathbb{N}}$-enriched category $\mathcal{C}$ with products, powers, and copowers. A $\mathcal{T}$-model in $\mathcal{C}$ comprises an object $X \in \mathcal{C}$ together with, for each $\mathrm{O} : (p \mid m_1 \ldots m_k)$, a morphism $\mathbf{y}(p) \cdot \big([\mathbf{y}(m_1), X] \times \cdots \times [\mathbf{y}(m_k), X]\big) \to X$, making a diagram commute for each equation in $\mathcal{T}$. (Here, we write $\mathbf{y}(m) := \mathbb{N}(m, -)$, and $(A \cdot X)$ and $[A, X]$ for the copower and power.) The elementary notion of model (Def. 6) is recovered because, for the symmetric monoidal closed structure on $\mathbf{Set}^{\mathbb{N}}$ itself, $([\mathbf{y}(m), X])(n) = X(n+m)$. This also connects with (3), since $(\triangleright X) = \mathbf{y}(1) \otimes X$ and $(\triangleleft X) = [\mathbf{y}(1), X]$.

4.2 Free models and monads

Strong monads are of fundamental importance to computational effects [21]. Algebraic theories give rise to strong monads via free models.

In slightly more detail, there is an evident notion of homomorphism applicable to Σ-structures and $\mathcal{T}$-models, and thus we can sensibly discuss Σ-structures and $\mathcal{T}$-models that are *free* over some collection $X \in \mathbf{Set}^{\mathbb{N}}$ of generators.

Informally, for a theory $\mathcal{T}$ we define $F_{\mathcal{T}}X \in \mathbf{Set}^{\mathbb{N}}$ by taking $F_{\mathcal{T}}X(n)$ to be the set of $=_{\mathcal{T}}$-equivalence classes of terms with parameter context $a_1, \ldots, a_n$ whose m_i-ary computation variables come from $X(m_i)$. More formally, we let

$$F_{\mathcal{T}}X(n) = \{\langle [x_1 : m_1, \ldots, x_k : m_k \mid a_1, \ldots a_n \vdash t]_{=_{\mathcal{T}}}, c_1, \ldots, c_k\rangle \mid c_i \in X(m_i)\}/\sim$$

where the equivalence relation $\sim$ allows us to α-rename context variables in the term judgements and apply permutation, contraction or weakening to the computation context paired with the corresponding transformation of the tuple $c_1, \ldots, c_k$. It is straightforward to make $F_{\mathcal{T}}X$ into a Σ-structure.

Proposition 3.

1. *$F_{\mathcal{T}}X$ is a $\mathcal{T}$-model, and moreover a free $\mathcal{T}$-model over X.*
2. *$F_{\mathcal{T}}$ extends to a monad on $\mathbf{Set}^{\mathbb{N}}$, strong for the Day tensor.*
3. *The derivable equality $(=_{\mathcal{T}})$ in a parameterized algebraic theory $\mathcal{T}$ is complete: if an equation is valid in every $\mathcal{T}$-model, then it is derivable in $=_{\mathcal{T}}$.*

A monad T on $\mathbf{Set}^{\mathbb{N}}$ strong for the Day tensor is a monad in the usual sense equipped with a strength $X \otimes TY \to T(X \otimes Y)$, where $\otimes$ is the Day tensor defined in Rem. 4.

Proof (notes). For (3), the monadic unit introduces variables and the bind is substitution. (In fact, this is part of an equivalence between such sifted-colimit-preserving strong monads and parameterized theories, e.g. [38, §5].)

Below we will consider explicit syntax-free characterizations of the free models for particular scoped theories.

In the case of a theory without equations, we recover exactly the scoped monad of Prop. 1 that was first given in [23]:

Theorem 1. *Consider signatures for algebraic Σ and scoped Σ' effects with no equations, inducing a parameterized algebraic theory $\mathcal{T}$ (via Rem. 2). We have an isomorphism of monads $F_{\mathcal{T}} \cong \left(\bar{\Sigma} + \left(\bar{\Sigma'} \circ \triangleleft\right) + \triangleright\right)^*$.*

Proof (notes). To see this, we use the description of $F_{\mathcal{T}}X(n)$ as a set of equivalence classes of $\mathcal{T}$-terms with computation variables coming from X. Consider the outermost operation of such a term: each of $\bar{\Sigma}$, $\left(\bar{\Sigma'} \circ \triangleleft\right)$ and $\triangleright$ on the right-hand-side corresponds to one of the three possibilities for this operation, algebraic, scoped or close respectively. Scoped operations bind a parameter and close consumes a parameter, hence the need for $\triangleleft/\triangleright$ on the right-hand-side: $\triangleleft$ increases the index n by 1 and $\triangleright$ decreases it, in keeping with Def. 5. Both $\triangleleft/\triangleright$ are characterized in Rem. 4 in terms of the Day tensor of $\mathbf{Set}^{\mathbb{N}}$.

4.3 Free models for scoped effects

We now turn to some concrete models from [23]. To characterize them as certain free models of parameterized algebraic theories, we need the following notion.

Definition 7. *$X \in \mathbf{Set}^{\mathbb{N}}$ is* truncated *if $X(n+1) = \emptyset$ for all $n \in \mathbb{N}$.*

Equivalently, X is truncated if $X = \upharpoonright(X(0))$. The free model on a truncated X corresponds to the case where computation variables can only denote programs with no open scopes. This is the case in the development of [23], where if the programmer opens a scope, a matching closing of the scope is implicitly part of the program.

Nondeterminism. Recall the parameterized theory for nondeterminism with once (signature in Ex. 6 and equations in Fig. 2). It follows from Prop. 3 that this theory has a free model on each X in $\mathbf{Set}^{\mathbb{N}}$, with carrier denoted by $T_{\mathbf{o}}(X) \in \mathbf{Set}^{\mathbb{N}}$. For X truncated, the free model on X has an elegant description:

$$T_{\mathbf{o}}(X)(n) \cong \mathsf{List}^{n+1}(X(0)).$$

In this case the interpretation of once chooses the first element of a list and closing a scope wraps its continuation as a singleton list. Choice is interpreted as list concatenation ($+\!\!+$), and failure as the empty list ($[]$):

$$\begin{array}{ll}
\mathsf{once}_n : T_{\mathbf{o}}(X)(n+1) \to T_{\mathbf{o}}(X)(n) & \mathsf{once}_n([]) = [],\ \mathsf{once}_n([x, \ldots]) = x \\
\mathsf{close}_n : T_{\mathbf{o}}(X)(n) \to T_{\mathbf{o}}(X)(n+1) & \mathsf{close}_n(x) = [x] \\
\mathsf{or}_n : T_{\mathbf{o}}(X)(n) \times T_{\mathbf{o}}(X)(n) \to T_{\mathbf{o}}(X)(n) & \mathsf{or}_n(x_1, x_2) = x_1 +\!\!+\, x_2 \\
\mathsf{fail}_n : 1 \to T_{\mathbf{o}}(X)(n) & \mathsf{fail}_n() = []
\end{array}$$

In fact the model $T_{\mathbf{o}}(X)$ we just described is the same as the model for nondeterminism from [23, Ex. 4.2]:

Theorem 2. *The model for nondeterminism with* once *from [23, Ex. 4.2], starting from a set A, is the free model on $\upharpoonright A \in \mathbf{Set}^{\mathbb{N}}$ for the parameterized theory of nondeterminism with* once *(Fig. 2).*

Proof (notes). We obtain a description of the free model by directing the equations from Fig. 2 and computing the normal forms. Then we specialize to $\upharpoonright A$.

Exceptions. Recall the parameterized theory of throwing and catching exceptions introduced in Ex. 7 and (9–11). For truncated $X \in \mathbf{Set}^{\mathbb{N}}$, the free model of the theory of exceptions has carrier:

$$T_{\mathbf{c}}(X)(n) = X(0) + \{e_0, \dots, e_n\}$$

where e_{n-i} corresponds to the term (in normal form) that closes i scopes then throws.

To define the operations catch_n and close_n we pattern match on the elements of $T_{\mathbf{c}}(X)(n+1)$ using the isomorphism $T_{\mathbf{c}}(X)(n+1) \cong T_{\mathbf{c}}(X)(n) + \{e_{n+1}\}$. Below, x is an element of $T_{\mathbf{c}}(X)(n)$, standing for a computation in normal form:

$$\begin{aligned}
&\mathsf{catch}_n : T_{\mathbf{c}}(X)(n+1) \times T_{\mathbf{c}}(X)(n+1) \to T_{\mathbf{c}}(X)(n)\\
&\mathsf{catch}_n(x, -) = x,\ \ \mathsf{catch}_n(e_{n+1},\, x) = x,\ \ \mathsf{catch}_n(e_{n+1},\, e_{n+1}) = e_n
\end{aligned}$$

$$\begin{aligned}
&\mathsf{close}_n : T_{\mathbf{c}}(X)(n) \to T_{\mathbf{c}}(X)(n+1) && \mathsf{close}_n(x) = x\\
&\mathsf{throw}_n : 1 \to T_{\mathbf{c}}(X)(n) && \mathsf{throw}_n() = e_n
\end{aligned}$$

The cases in the definition of catch_n correspond to equations (11), (9), (10) respectively. In the third case, an exception inside $n+1$ scopes in the second argument of catch becomes an exception inside n scopes.

Theorem 3. *The model for exception catching from [23, Ex. 4.5], which starts from a set A, is the free model on $\restriction A \in \mathbf{Set}^{\mathbb{N}}$ for the parameterized theory of exceptions (9–11).*

State with local values. Recall the parameterized theory of mutable state with local values in Ex. 8 and its equations (12–17). The free model, in the sense of Prop. 3, on a truncated $X \in \mathbf{Set}^{\mathbb{N}}$ has carrier:

$$T_{\mathbf{l}}(X)(0) = \mathbb{2} \Rightarrow X(0) \times \mathbb{2} \qquad\qquad T_{\mathbf{l}}(X)(n+1) = \mathbb{2} \Rightarrow T_{\mathbf{l}}(X)(n)$$

The operations on this model are

$$\begin{aligned}
&\mathsf{local}^i_n : T_{\mathbf{l}}(X)(n+1) \to T_{\mathbf{l}}(X)(n) && \mathsf{local}^i_n(f) = (f\, i)\\
&\mathsf{close}_n : T_{\mathbf{l}}(X)(n) \to T_{\mathbf{l}}(X)(n+1) && \mathsf{close}_n(f) = \lambda s.\, f\\
&\mathsf{put}^i_n : T_{\mathbf{l}}(X)(n) \to T_{\mathbf{l}}(X)(n) && \mathsf{put}^i_n(f) = \lambda s.\, f\, i\\
&\mathsf{get}_n : T_{\mathbf{l}}(X)(n)^2 \to T_{\mathbf{l}}(X)(n) && \mathsf{get}_n(f,g) = \lambda s. \begin{cases} f\ s & s = 0\\ g\ s & \text{otherwise}\end{cases}
\end{aligned}$$

Notice that the continuation of local^i uses the new state i, whereas close discards the state s which comes from the scope that is being closed.

If we only consider equations (12–16), omitting (17), the carrier of the free model on a truncated $X \in \mathbf{Set}^{\mathbb{N}}$ is:

$$T'_{\mathbf{l}}(X)(0) = \mathbb{2} \Rightarrow X(0) \times \mathbb{2} = T_{\mathbf{l}}(X)(0), \quad T'_{\mathbf{l}}(X)(n+1) = \mathbb{2} \Rightarrow T'_{\mathbf{l}}(X)(n) \times \mathbb{2}$$

In fact, $T'_{\mathbf{l}}(X)$ is the model of state with local proposed in [23, §7.1]:

Theorem 4. *Consider the example of state with local variables from [23], specialized to one memory location storing one bit, reading the return type a as a set A. The model proposed in [23, §7.1] is the free model on $\upharpoonright A$ for the parameterized algebraic theory with equations 12–16.*

The interpretations in $T_1(X)$ and $T_1'(X)$ (i.e that of [23]) of programs with no open scopes agree:

Proposition 4. *Consider a fixed context of computation variables $\Gamma = (x_1 : 0, \dots, x_n : 0)$ and a truncated $X \in \mathbf{Set}^{\mathbb{N}}$. For any term $\Gamma \mid - \vdash t$, the following two interpretations coincide at index* 0*:*

$$[\![t]\!]_{T_1(X),0} = [\![t]\!]_{T_1'(X),0} : T_1(X)(0)^n \to T_1(X)(0),$$

under the identification $T_1(X)(0) = T_1'(X)(0)$.

The restrictions of Γ to computation variables that do not depend on parameters and of Δ to be empty are reasonable because in the framework of [23], only programs with no open scopes are well-formed. Therefore, only such programs can be substituted in t, justifying the restriction of $[\![t]\!]_{T_1'(X)}$ to index 0.

5 Summary and research directions

We have provided a fresh perspective on scoped effects in terms of the formalism of parameterized algebraic theories, using the idea that scopes are resources (Rem. 2). As parameterized algebraic theories have a sound and complete algebraic theory (Props. 2, 3), this carries over to a sound and complete equational theory for scoped effects. We showed that our fresh perspective recovers the earlier models for scoped non-determinism, exceptions, and state (Thms. 2–4).

Here we have focused on equational theories for effects alone. But as is standard with algebraic effects, it is easy to add function types, inductive types, and so on, together with standard beta/eta theories (e.g. [25],[38, §5]). This can be shown sound by the simple models considered here, as indeed the canonical model $\mathbf{Set}^{\mathbb{N}}$ is closed and has limits and colimits.

Our fresh perspective opens up new directions for scoped effects, in theory and in practice. By varying the substructural laws of parameterized algebraic theories, we can recover foundations for scoped effects where scopes (as resources) can be reordered or discarded, i.e. where they are not well-bracketed, already considered briefly in the literature [23]. For example, the parameterized algebraic theory of qubits [38] might be regarded as a scoped effect, where we open a scope when a qubit is allocated and close the scope when it is discarded; this generalizes traditional scoped effects as multi-qubit operations affect multiple scopes.

Acknowledgements. We are grateful to many colleagues for helpful discussions and to the anonymous reviewers for their helpful comments and suggestions. This work was supported by the UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (reference number MR/T043830/1), ERC Project BLAST, and AFOSR Award No. FA9550–21–1–003.

References

1. Abramsky, S., Jung, A.: Domain theory. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. 3 (1994)
2. Bauer, A.: What is algebraic about algebraic effects and handlers? (2019). https://doi.org/10.48550/arXiv.1807.05923
3. Benton, N., Wadler, P.: Linear logic, monads and the lambda calculus. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science. pp. 420–431 (1996). https://doi.org/10.1109/LICS.1996.561458
4. Bizjak, A., Grathwohl, H.B., Clouston, R., Møgelberg, R.E., Birkedal, L.: Guarded dependent type theory with coinductive types. In: Jacobs, B., Löding, C. (eds.) Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science, vol. 9634, p. 20–35. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_2
5. Bosman, R., van den Berg, B., Tang, W., Schrijvers, T.: A calculus for scoped effects & handlers (2023). https://doi.org/10.48550/arXiv.2304.09697
6. Day, B.: On closed categories of functors. In: MacLane, S., Applegate, H., Barr, M., Day, B., Dubuc, E., Phreilambud, Pultr, A., Street, R., Tierney, M., Swierczkowski, S. (eds.) Reports of the Midwest Category Seminar IV. pp. 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg (1970)
7. Fiore, M.P., Staton, S.: Substitution, jumps, and algebraic effects. In: Proc. CSL-LICS2014 (2014)
8. Fiore, M., Szamozvancev, D.: Formal metatheory of second-order abstract syntax. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022). https://doi.org/10.1145/3498715, https://doi.org/10.1145/3498715
9. Fiore, M.P., Hur, C.: Second-order equational logic (extended abstract). In: Dawar, A., Veith, H. (eds.) Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6247, pp. 320–335. Springer (2010). https://doi.org/10.1007/978-3-642-15205-4_26, https://doi.org/10.1007/978-3-642-15205-4_26
10. Fiore, M.P., Mahmoud, O.: Second-order algebraic theories - (extended abstract). In: Hlinený, P., Kucera, A. (eds.) Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6281, pp. 368–380. Springer (2010). https://doi.org/10.1007/978-3-642-15155-2_33, https://doi.org/10.1007/978-3-642-15155-2_33
11. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)
12. Hyland, M., Plotkin, G., Power, J.: Combining effects: Sum and tensor. Theor. Comput. Sci. **357**(1), 70–99 (Jul 2006). https://doi.org/10.1016/j.tcs.2006.03.013
13. Katsumata, S.y., McDermott, D., Uustalu, T., Wu, N.: Flexible presentations of graded monads. Proc. ACM Program. Lang. **6**(ICFP) (aug 2022). https://doi.org/10.1145/3547654, https://doi.org/10.1145/3547654
14. Kelly, G., Power, A.: Adjunctions whose counits are coequalizers, and presentations of finitary enriched monads. Journal of Pure and Applied Algebra **89**(1), 163–179 (1993). https://doi.org/https://doi.org/10.1016/0022-4049(93)90092-8, https://www.sciencedirect.com/science/article/pii/0022404993900928
15. Lawvere, F.W.: Functorial semantics of algebraic theories. Proceedings of the National Academy of Sciences **50**(5), 869–872 (1963). https://doi.org/10.1073/pnas.50.5.869

16. Linton, F.E.J.: Some aspects of equational categories. In: Eilenberg, S., Harrison, D.K., MacLane, S., Röhrl, H. (eds.) Proceedings of the Conference on Categorical Algebra. pp. 84–94. Springer Berlin Heidelberg, Berlin, Heidelberg (1966). https://doi.org/10.1007/978-3-642-99902-4_3
17. Melliès, P.A.: Local states in string diagrams. In: Dowek, G. (ed.) Rewriting and Typed Lambda Calculi. pp. 334–348. Springer International Publishing, Cham (2014)
18. Melliès, P.A.: Segal condition meets computational effects. In: 2010 25th Annual IEEE Symposium on Logic in Computer Science. pp. 150–159 (2010). https://doi.org/10.1109/LICS.2010.46
19. Milner, R.: Communicating and Mobile Systems: The π-calculus. Cambridge University Press, United States (May 1999)
20. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings. Fourth Annual Symposium on Logic in Computer Science. pp. 14–23 (1989). https://doi.org/10.1109/LICS.1989.39155
21. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55 – 92 (1991). https://doi.org/https://doi.org/10.1016/0890-5401(91)90052-4, selections from 1989 IEEE Symposium on Logic in Computer Science
22. Petersen, L., Harper, R., Crary, K., Pfenning, F.: A type theory for memory allocation and data layout. In: POPL 2003 (2003)
23. Piróg, M., Schrijvers, T., Wu, N., Jaskelioff, M.: Syntax and semantics for operations with scopes. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. p. 809–818. LICS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3209108.3209166
24. Pitts, A.M.: Structural recursion with locally scoped names. Journal of Functional Programming **21**(3), 235–286 (2011). https://doi.org/10.1017/S0956796811000116
25. Plotkin, G., Power, J.: Algebraic operations and generic effects. Applied Categorical Structures **11**, 69–94 (2003)
26. Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) Foundations of Software Science and Computation Structures, 5th International Conference. pp. 342–356. FOSSACS 2002, Springer (2002), https://doi.org/10.1007/3-540-45931-6_24
27. Plotkin, G., Power, J.: Computational effects and operations: An overview. Electr. Notes Theor. Comput. Sci. **73**, 149–163 (10 2004). https://doi.org/10.1016/j.entcs.2004.08.008
28. Plotkin, G., Pretnar, M.: Handling algebraic effects. Logical Methods in Computer Science **9**(4) (Dec 2013). https://doi.org/10.2168/lmcs-9(4:23)2013
29. Plotkin, G.D., Power, J.: Adequacy for algebraic effects. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. Lecture Notes in Computer Science, vol. 2030, pp. 1–24. Springer (2001), https://doi.org/10.1007/3-540-45315-6_1
30. Polakow, J.: Ordered linear logic and applications. Ph.D. thesis, USA (2001)
31. Power, J.: Enriched Lawvere theories. Theory and Applications of Categories **6**(7), 83–93 (1999)
32. Power, J.: Semantics for local computational effects. Electronic Notes in Theoretical Computer Science **158**, 355–371 (May 2006). https://doi.org/10.1016/j.entcs.2006.04.018
33. Stark, I.: Free-algebra models for the pi -calculus. Theor. Comput. Sci. **390**(2-3), 248–270 (2008). https://doi.org/10.1016/j.tcs.2007.09.024, https://doi.org/10.1016/j.tcs.2007.09.024

34. Staton, S., Stein, D., Yang, H., Ackerman, N.L., Freer, C., Roy, D.: The Beta-Bernoulli process and algebraic effects. In: Proc. ICALP 2018 (2018)
35. Staton, S.: An algebraic presentation of predicate logic - (extended abstract). In: FOSSACS 2013 (2013)
36. Staton, S.: Instances of computational effects: An algebraic perspective. In: LICS 2013 (2013)
37. Staton, S.: Freyd categories are enriched lawvere theories. Electronic Notes in Theoretical Computer Science **303**, 197–206 (2014). https://doi.org/https://doi.org/10.1016/j.entcs.2014.02.010, https://www.sciencedirect.com/science/article/pii/S157106611400036X, proceedings of the Workshop on Algebra, Coalgebra and Topology (WACT 2013)
38. Staton, S.: Algebraic effects, linearity, and quantum programming languages. In: POPL 2015 (2015)
39. Thomson, P., Rix, R., Wu, N., Schrijvers, T.: Fusing industry and academia at GitHub (experience report). Proc. ACM Program. Lang. **6**(ICFP) (aug 2022). https://doi.org/10.1145/3547639, https://doi.org/10.1145/3547639
40. van den Berg, B., Schrijvers, T.: A framework for higher-order effects & handlers (2023). https://doi.org/10.48550/arXiv.2302.01415
41. Wu, N., Schrijvers, T., Hinze, R.: Effect Handlers in Scope. pp. 1–12 (2014). https://doi.org/10.1145/2633357.2633358
42. Yang, Z., Paviotti, M., Wu, N., van den Berg, B., Schrijvers, T.: Structured handling of scoped effects. p. 462–491. Springer-Verlag, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-030-99336-8_17
43. Yang, Z., Wu, N.: Modular models of monoids with operations. Proc. ACM Program. Lang. **7**(ICFP) (aug 2023). https://doi.org/10.1145/3607850

Monadic Intersection Types, Relationally

Francesco Gavazzo[1], Riccardo Treglia[2], and Gabriele Vanoni[3](✉)

[1] Università di Padova, Padova, Italy
francesco.gavazzo@unipd.it

[2] King's College London, London, UK
riccardo.treglia@kcl.ac.uk

[3] IRIF, CNRS, Université Paris Cité, Paris, France
gabriele.vanoni@irif.fr

Abstract. We extend intersection types to a computational λ-calculus with algebraic operations *à la* Plotkin and Power. We achieve this by considering monadic intersections—whereby computational effects appear not only in the operational semantics, but also in the *type system*. Since in the effectful setting termination is not anymore the only property of interest, we want to analyze the interactive behavior of typed programs with the environment. Indeed, our type system is able to characterize the natural notion of observation, both in the finite and in the infinitary setting, and for a wide class of effects, such as output, cost, pure and probabilistic nondeterminism, and combinations thereof. The main technical tool is a novel combination of syntactic techniques with abstract relational reasoning, which allows us to lift all the required notions, e.g. of typability and logical relation, to the monadic setting.

1 Introduction

Type systems are a key aspect of programming languages, ensuring good behavior during the execution of programs, such as absence of errors, termination, or properties such as productivity, safety, and reachability. Additionally, they ensure it in a *compositional* way, that is, if programs are assembled according to the underlying type discipline, then the good behavior is ensured also for the composed program.

Intersection Types. Type systems have solid roots in logic and proof theory, as witnessed by the Curry-Howard correspondence between simple types and intuitionistic natural deduction. However, in the theory of the λ-calculus, there is another use of types that has been studied at length: *intersection types*. They were introduced by Coppo and Dezani-Ciancaglini in the late 70's [16] to overcome the limitations of Curry's type discipline and enlarge the class of terms that can be typed. This is reached by means of a new type constructor, the *intersection*. In this way, one can assign a finite set of types to a term, thus providing a form of finite polymorphism.

Similarly to simple types, intersection types ensure termination. In contrast to most notions of types, however, they also *characterize* termination, that is, they type *all* terminating λ-terms. They can be seen as a compositional way of defining operational semantics, or, in a dual way, as a syntactic presentation of

S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 22–51, 2024.
https://doi.org/10.1007/978-3-031-57262-3_2

denotational models. Additionally, intersection types have shown to be remarkably flexible, since different termination forms can be characterized by tuning details of the type system (*e.g.*, weak/strong full normalization, head/weak/call-by-value evaluation). Termination being only *semi-decidable*, type inference cannot be decidable, which is why standard intersection types are somewhat incompatible with programming practice, although some restricted forms of intersection types have found applications in programming, see for example [33,53,34,28], or the recent survey by Bono and Dezani-Ciancaglini [12].

Beyond the Pure λ-Calculus. Intersection types have been mostly developed in the realm of the pure λ-calculus. However, current programming languages are deeply effectful, exhibiting several simultaneous impure behaviours, such as raising exceptions, performing input/output operations, sampling from distributions, etc. Reasoning about effectful programs becomes a challenging goal since their behaviour becomes highly interactive, depending on the external environment. Type-based techniques seem very interesting in this respect, since they enable *modular* and *compositional* analysis of program behaviour. In particular, intersection type systems have already been successfully adapted to some concrete computational effects, such as probabilistic [13,18], and pure nondeterminism [65]. In spite of the remarkable results achieved by each of these formalisms – for instance, probabilistic intersection types have been proved to characterize almost-sure termination – all of these come with a major drawback: they are tailored to the specific family of effects considered. This results in a lack of robustness and modularity when it comes to extending languages with new effects. For instance, probabilistic intersection types as they are, cannot cope with, *e.g.*, languages with *both* randomness and output. This problem can be fixed (output is a well-behaved effect that nicely interacts with probabilistic nondeterminism) but in highly non-modular way. One has, in fact, to re-engineer the whole theoretical framework behind probabilistic intersection types to account for output, too.

Now, the leading question is: *can intersection types be scaled up in the case of effectful λ-calculi, in a modular way?* In this paper, we answer this question in the affirmative by developing a general *monadic intersection type system* for an untyped computational λ-calculus [50,51] with algebraic operations *à la* Plotkin and Power [57]. In fact, our system covers both finitary and infinitary effectful operational behaviours in a sound and complete way, and generalises existing effectful intersection type systems, such as probabilistic intersection types. To achieve this result, we combine state-of-the-art techniques in monadic semantics, intersection types, and relational reasoning, in a novel and nontrivial way.

Monadic Semantics. As we have already mentioned, our work is grounded on the theoretical framework pioneered by Moggi [50,51], which describes computational effects via monads [49]. Moggi's work described how monads could give denotational semantics to effectful programs, but did not tell anything about how computational effects are actually produced. Plotkin and Power [57,58] introduced *algebraic effects* as a way of giving monadic semantics, in the style of Moggi, to certain operations which actually produce computational side effects. The core syntax of effectful languages can be thus given in terms of computational calculi with effect-triggering operations. But what about the operational semantics and, most importantly, the (intersection) type system?

There is a well-known way to give effectful operational semantics to programming languages, namely making operational semantics effectful itself. That is, if we model the operational semantics of a language using a transition relation between terms, then we can make such a relation monadic by relating terms with *monadic terms*, i.e. terms encapsulated by a monad, encoding the effects produced during the computation. Such relations—i.e. relations of the form $R \subseteq A \times T(B)$, with T a monad—are known as *monadic* or *Kleisli relations* and have been successfully used to give operational semantics to monadic calculi [36].

What about the type system? In its bare essence, it is given by a relation between terms and types, hence leading to a situation similar to the one of operational semantics. The analogy is no coincidence: as we obtain monadic operational semantics relying on the theory of monadic relations, the very same theory allows us to define monadic type systems: a monadic typing relation associates terms with *monadic types*. This idea has already been exploited by Dal Lago and collaborators [13,18], who realised that to extend intersection types to probabilistic languages one has to type expressions not with types, but with *distribution of types*.[4] This is nothing but a monadic typing relation instantiated to the distribution monad.

Relational Reasoning. Working at the abstract level of monadic relations gives several advantages, both in terms of modularity and expressiveness. Concerning the former, we shall develop abstract proof techniques that allow us to give proofs of subject reduction and expansion independently of the underlying monad. Such factorization results rely on the theory of (monadic) relational extensions. Here, one studies how to extend a monadic relation $R \subseteq A \times T(B)$, such as the one modeling one-step operational semantics, or typing between terms and monadic types, to a relation $R^{\dagger} \subseteq T(A) \times T(B)$, necessary to model operational semantics, or typing, of *monadic* terms. Such extensions, even if canonical, do not exist in general, and a celebrated result by Barr [10] gives necessary and sufficient conditions for the existence of relational extensions: monads must be *weakly cartesian* [71,15]. Intuitively, being weakly cartesian means that during the evaluation there is no loss of information about the performed effects. This is a form of reversibility, that is needed, *e.g.*, in subject expansion,

This restriction rules out from our analysis monads such as the powerset or the distribution monad. Although that may appear as a weakness of our framework, it actually exploits a nontrivial limitative result that has already been observed in different forms in the literature: results involving forms of reversibility, such as subject expansion, are simply not available when monads are not weakly

[4] Actually, as we will show in Section 4, distributions do not behave well in this case. This is why in [13] convex sets of distributions are used. Multidistributions are instead used in [18].

cartesian. This is the deep reason why probabilistic intersection types are defined via the multi-distribution, rather than distribution, monad. Remarkably, the same kind of limitative result has also been proved in the setting of monadic operational semantics and rewriting [36].

Contributions. In this paper, we introduce the first (to the best of our knowledge) intersection type system handling the computational λ-calculus with algebraic operations. This is done by letting not only terms, but also intersection types be monadic. The main idea of intersection types is that they are a static way to describe the mechanism of evaluation of programs. Since the operational semantics of effectful languages can be conveniently described by evaluation inside monads, it is natural to embed also intersection types inside them. This way, we are able to push forward the correspondence between intersection type derivations and term evaluation to the effectful/monadic setting. More precisely, we develop several contributions and theoretical advances:

- *The Type System:* We provide the first idempotent[5] intersection type system for a λ-calculus with algebraic operations which is parametric in the underlying monad. We design the type system in such a way that not only terms, but also types become monadic.
- *Characterization of Observable Behavior:* Differently from the pure untyped setting, in which intersection types characterize (different forms of) termination of programs, in the effectful setting we would like to characterize via the type system *all* the effects produced during the evaluation. Indeed, we obtain such a result by generalizing standard soundness and completeness theorems, via abstract relational techniques. In particular, observable behaviors of typable (*i.e.* all the terminating) terms can be read out of their types.
- *Intrinsic Limits:* Our approach comes with the already described intrinsic limits about the class of well-behaving monads (the weakly cartesian ones). Moreover, if one sticks with the finitary case, where the natural notion of convergence is must termination, another restriction on the kinds of admissible operations is needed. Indeed, also operations that erase arguments break the subject expansion, and thus the completeness of the system. Still, this restriction can be removed considering an infinitary semantics.
- *The Infinitary Case:* Some interesting notions of observation, such as the probability of convergence in probabilistic calculi, are naturally infinitary. For this reason, we extend our type system to capture infinitary behaviors. Interestingly, we need to add just one typing rule to the previous (finitary) system, namely the one that can type every term with the bottom of the monad.

[5] The choice of the idempotent variant of intersection types should not be taken too strictly. All the results of the paper hold also turning intersections into multisets, *mutatis mutandis*. Moreover, the meta-theory in the idempotent case is more involved (requiring logical relations to prove soundness), and we show this way the strength of our approach. Still, it is not an exercise of style, because intersection type systems used to formalize higher-order model checking algorithms (see Sec. 7 for a more detailed discussion) *must* be idempotent, since otherwise one would lose decidability.

Naturally, this extension requires the monads to satisfy more conditions (mainly domain theoretic ones). Remarkably, this way we are actually able to relax the constraint on non-erasing operations introduced in the finitary system.

Related Work. To the best of our knowledge, this is the first work about monadic intersection type systems for effectful calculi with algebraic operations, which are parametric on the underlying monad. On an orthogonal axis, an intersection type system for a variant over Moggi's computational λ-calculus, but without any reference to algebraic operations, has been proposed in [25], while intersections types have developed for calculi with continuations in [9] and, paired with union types, in [45]. With concrete monads, instead, various proposals have appeared for the state monad [29,26,27,5], and the distribution monad [31,13,18]. Moreover, lifting the monad to the type system has already been done in a series of works by Dal Lago and coauthors, *e.g.* to analyze complexity [6,22] and recently for the state monad in [5]. More on the programming side, intersection types have been proposed for a λ-calculus with computational side effects and reference types in [23], but without any reference to monads.

Proofs. Omitted proofs are in the technical report [37].

2 Intersection Types and the CbV λ-Calculus

We devote this section to a gentle introduction to intersection type systems. For the moment, we do not consider effectful calculi and we set our analysis in the (almost) standard setting of Plotkin's call-by-value (CbV) λ-calculus [55]. Actually, the calculus we present is the kernel of CbV λ-calculus, that is as expressive as the Plotkin's [2,32], but allows only a restricted form of term application.

The (kernel) CbV λ-Calculus. Given a countable set of variables $\mathcal{V}$, values and computations are defined by mutual induction as follows:

$$\begin{aligned} \textsc{Computations}\ \mathbb{C} \ni t, u &::= v \mid vt \\ \textsc{Values}\ \mathbb{V} \ni v, w &::= x \in \mathcal{V} \mid \lambda x.t \\ \textsc{Eval. Contexts}\ \mathbb{E} \ni E &::= [\cdot] \mid vE \end{aligned}$$

Free and *bound variables* are defined as usual: $\lambda x.t$ binds x in t. Terms are considered modulo α-equivalence. Capture-avoiding (meta-level) substitution of u for all the free occurrences of x in t is written $t\{x/u\}$. As it is customary when working in the CbV setting, we restrict ourselves to *closed* terms, *i.e.*, we consider only terms without free variables. This means that the normal forms are all and only the closed values, noted $\mathbb{V}^\bullet$, *i.e.* the λ-abstractions. The traditional β rule is restricted to values, *i.e.* only (closed) values can be substituted: $(\lambda x.t)v \rightarrowtail t\{x/v\}$.

$$\frac{A \in I}{\Gamma, x : I \vdash x : A}\ \text{VAR} \qquad \frac{[\Gamma \vdash v : A_i]_{i \in F}}{\Gamma \vdash v : \{A_i\}_{i \in F}}\ \text{INT}$$

$$\frac{\Gamma, x : I \vdash t : J}{\Gamma \vdash \lambda x.t : I \to J}\ \text{ABS} \qquad \frac{\Gamma \vdash v : I \to J \quad \Gamma \vdash t : I}{\Gamma \vdash vt : J}\ \text{APP}$$

Fig. 1: The intersection type system for closed call-by-value.

The deterministic operational semantics $\mapsto$ is obtained by closing the β rule (by value) $\rightarrowtail$ w.r.t. all evaluation contexts. Please notice that although we restricted term application to have a value as the left subterm, we can recover the usual application between terms as $tu := (\lambda x.xu)t$, where x is a fresh variable.

Intersection Types for the CbV λ-Calculus. The CbV λ-Calculus is a universal, *i.e.* Turing complete, model of computation. This way its halting problem is obviously undecidable. Nonetheless, terminating terms can be characterized by syntactic means. Intersection types are one way of doing this, in a compositional and logical way. The grammar of types is reminiscent of the call-by-value translation $(\cdot)^{\mathsf{v}}$ of intuitionistic logic into linear logic [38,30] $(A \to B)^{\mathsf{v}} = !(A^{\mathsf{v}}) \multimap !(B^{\mathsf{v}})$. Semantically, they can be seen as a syntactical presentation of filter models of the λ-calculus [17]. The grammar for types is based on *two* layers of types, defined in a mutually recursive way, *value* types A, and *intersections* (*i.e.* sets) I of value types.

$$\begin{aligned} \textsc{Value Types}\ \mathbb{A} \ni A &::= I \to J \\ \textsc{Intersections}\ \mathbb{I} \ni I, J &::= \{A_1, \dots, A_n\} \quad n \geq 0 \\ \textsc{Types}\ \mathbb{G} \ni G &::= A \mid I \end{aligned}$$

Remark 1. Please notice that intersections can be empty. The empty intersection type $\mathbf{0} := \{\}$ stands for the type of *erasable* terms, which in Closed CbV are just those terms evaluating to closed values (*i.e.* λ-abstractions). In CbV, terminating terms and erasable terms coincide, as the argument of a β-reduction has to be evaluated before being erased (and so its evaluation has to terminate).

Type environments, ranged over by Γ, Δ, are total maps from variables to intersection types such that only finitely many variables are mapped to non-empty intersection types, and we write $\Gamma = x_1 : I_1, \dots, x_n : I_n$ if $\mathsf{dom}(\Gamma) = \{x_1, \dots, x_n\}$. Type judgments have the form $\Gamma \vdash t : G$. The typing rules are in Fig. 1, where F stands for a finite, possibly empty, set of indexes; type derivations are written π and we write $\pi \triangleright \Gamma \vdash t : G$ for a type derivation π with the judgment $\Gamma \vdash t : G$ as its conclusion.

Intuitively, intersections are needed because, during the evaluation of a term t, a subterm of t can assume different types. For example in $(\lambda x.xx)(\lambda y.y)$, the argument $\lambda y.y$ has type $\mathbf{0} \to \mathbf{0}$, when it substitutes the first occurrence of x in functional position, and has type $\mathbf{0}$ when it substitutes the second occurrence of x in argument position. These different uses, which require different types, are encoded into the intersection type. Moreover, the type system, contrarily to

$$
\dfrac{
  \dfrac{
    \dfrac{}{x : \{\mathtt{id}\} \vdash x : \mathtt{id}}\,{\scriptstyle \mathrm{VAR}}
    \quad
    \dfrac{}{x : \{\mathtt{id}\} \vdash x : \mathbf{0}}\,{\scriptstyle \mathrm{INT}}
  }{x : \{\mathtt{id}\} \vdash xx : \mathbf{0}}\,{\scriptstyle \mathrm{APP}}
}{\vdash \lambda x.xx : \{\mathtt{id}\} \to \mathbf{0}}\,{\scriptstyle \mathrm{ABS}}
\qquad
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{y : \{\mathtt{id}\} \vdash y : \mathtt{id}}\,{\scriptstyle \mathrm{VAR}}
    }{y : \{\mathtt{id}\} \vdash y : \{\mathtt{id}\}}\,{\scriptstyle \mathrm{INT}}
  }{\vdash \lambda y.y : \{\mathtt{id}\} \to \{\mathtt{id}\}}\,{\scriptstyle \mathrm{ABS}}
  \quad
  \dfrac{
    \dfrac{
      \dfrac{}{z : \mathbf{0} \vdash z : \mathbf{0}}\,{\scriptstyle \mathrm{INT}}
    }{\vdash \lambda z.z : \mathtt{id}}\,{\scriptstyle \mathrm{ABS}}
  }{\vdash \lambda z.z : \{\mathtt{id}\}}\,{\scriptstyle \mathrm{INT}}
}{\vdash (\lambda y.y)(\lambda z.z) : \{\mathbf{0} \to \mathbf{0}\}}\,{\scriptstyle \mathrm{APP}}
$$
$$
\dfrac{\vdash \lambda x.xx : \{\mathtt{id}\} \to \mathbf{0} \qquad \vdash (\lambda y.y)(\lambda z.z) : \{\mathbf{0} \to \mathbf{0}\}}{\vdash (\lambda x.xx)(\mathsf{II}) : \mathbf{0}}\,{\scriptstyle \mathrm{APP}}
$$

Fig. 2: Type derivation for $\vdash (\lambda x.xx)(\mathsf{II}) : \mathbf{0}$. We set $\mathtt{id} := \mathbf{0} \to \mathbf{0}$.

what happens in call-by-name, and consistently with the rationale of CbV, needs arguments of applications to be typed (with an intersection) just once.

Example 1. We provide the type derivation for the term $\vdash (\lambda x.xx)(\mathsf{II}) : \mathbf{0}$, where $\mathsf{I} := \lambda x.x$, in Fig. 2. One can notice that our example term, being CbV-normalizing, can be typed with $\mathbf{0}$.

Characterization of Termination. Intersection types characterize Closed CbV termination, that is, they type all and only those λ-terms that terminate with respect to Closed CbV. We give a very brief overview of how this result is achieved. In the following sections, we shall prove all these results in the effectful setting. The reader could, however, benefit from the exposition of the main steps in this simpler setting.

Similarly to more traditional type systems, this intersection type system enjoys subject reduction, *i.e.* types are preserved under reduction.

Lemma 1 (Subject Reduction). *Let t be a closed λ-term. If $\vdash t : I$ and $t \mapsto u$, then $\vdash u : I$.*

Moreover, as with simple types, all typable terms terminate.

Proposition 1 (Soundness). *Let t be a closed λ-term. If $\vdash t : I$, then t has normal form.*

Contrarily to simple types, intersection types satisfy also subject *expansion*. This means that types are preserved also by backward reductions (*i.e.* expansions).

Lemma 2 (Subject Expansion). *Let t be a closed λ-term. If $\vdash u : I$ and $t \mapsto u$, then $\vdash t : I$.*

Together with the fact that normal forms, *i.e.* λ-abstractions, can always be typed with the empty type $\mathbf{0}$, this gives the completeness of the type system, *i.e.* the fact that every terminating term is typable.

Proposition 2 (Completeness). *Let t be a closed λ-term. If t has normal form, then there exists an intersection type I such that $\vdash t : I$.*

Putting soundness and completeness together, we obtain the full characterization of termination via typability.

Theorem 1 (Characterization). *Let t be a closed λ-term. Then there exists an intersection type I such that $\vdash t : I$ if and only if t has normal form.*

3 Preliminaries on Monads, Algebraic Effects, Operations

In this section, we recall some preliminary notions on monads [49], algebras [60], and relational reasoning [61], that will be central to the rest of this paper. Due to space constraints, there is no hope to be comprehensive, and thus we assume the reader to have minimal familiarity with those fields. Unless explicitly stated, we work in the category **Set** of sets and functions and we tacitly restrict all definitions to it. Since we will extensively work with relations, we employ the relational notation even for functions, writing $f; g : A \to C$ for the composition (in diagrammatic order) of $f : A \to B$ and $g : B \to C$, and $\mathbf{1}_A : A \to A$ (mostly omitting subscripts) for the identity function.

Monads and Algebraic Effects. We use monads [50,51] to model computational effects.

Definition 1 (Monad). *A* monad *(on* **Set***) is a triple* (T, η, μ) *consisting of a functor* T *(on* **Set***) together with two natural transformations:* $\eta : 1_{\mathbf{Set}} \Rightarrow T$ *(called* unit*) and* $\mu : TT \Rightarrow T$ *(called* multiplication*) subject to the following laws:* $\eta; \mu = T(\eta); \mu = \mathbf{1}$ *and* $T(\mu); \mu = \mu; \mu$*.*

Given a monad (T, η, μ) we oftentimes identify it with its carrier functor. Moreover, we write $f^\dagger : T(A) \to T(B)$ for the Kleisli extension of $f : A \to T(B)$, where $f^\dagger := T(f); \mu$, and $\gg\!=$ for the binding operator induced by $-^\dagger$. Such an operator maps a monadic element $t \in T(A)$ and a monadic function $f : A \to T(B)$ to the monadic element $t \gg\!= f$ in $T(B)$ defined as $f^\dagger(t)$. It is well-known that using this construction a monad could be presented also as a Kleisli triple $(T, \eta, -^\dagger)$, or with the bind operation instead of μ, *i.e.* as $(T, \eta, \gg\!=)$ [70].

To model how actual effects are produced, Plotkin and Power [54,58] introduced the notion of an *algebraic operation*, which we shall use to make calculi truly effectful.

Definition 2 (Algebraic Operation). *Given a monad* (T, η, μ)*, an* n*-ary* algebraic operation *is a natural transformation* $\alpha : T^n \Rightarrow T$ *respecting the monad multiplication.*

From an operational perspective, algebraic operations describe those operations whose execution is independent of the context in which they are executed.

Example 2 (Concrete Monads and Operations).

1. Divergent computations are modelled by the *maybe* or *partiality* monad $(\mathcal{E}, \eta, \mu)$, where $\mathcal{E}(A) := A + \{\bot\}$, η is the left injection ι_ℓ, and $\mu : ((A + \{\bot\}) + \{\bot\}) \to A + \{\bot\}$ sends $\iota_\ell(\iota_\ell(x))$ to $\iota_\ell(x)$, and all the rest to $\bot$. Therefore, an element in $\mathcal{M}A$ is either an element $a \in A$ (meaning that we have a terminating computation returning a), or the element $\bot$ (meaning that the computation diverges). As non-termination is an intrinsic feature of complete programming languages, we do not consider explicit operations to

produce divergence. Nonetheless, notice that we might consider the constant $\bot$ as a zero-ary operation producing divergence (linguistically, this essentially corresponds to adding an always diverging constant `diverge`).

2. Replacing $\{\bot\}$ with a set of errors *Err*, we obtain the exception monad. Exceptions are produced by means of 0-ary operations $\mathtt{raise}_e$ indexed by elements in *Err*.
3. Probabilistic computations are modelled by the (finitary) distribution monad $(\mathcal{D}, \eta, \mu)$, where $\mathcal{D}(X)$ is the set of distributions over X with *finite* support (where the support of a distribution $d \in \mathcal{D}(X)$ is defined as $\mathsf{supp}(d) := \{x \in X \text{ such that } d(x) > 0\}$), $\eta(x)$ is the Dirac distribution on x, and $\mu(\Phi)(x) := \sum_{\phi} \Phi(\phi) \cdot \phi(x)$. Finite distributions are produced using weighted sums, *i.e.* $[0,1]$-indexed binary operations $+_p$ defined thus: $(\phi_1 +_p \phi_2)(x) := p \cdot \phi_1(x) + (1-p) \cdot \phi_2(x)$. In a similar fashion, one defines the finitary subdistribution monad $\mathcal{D}^{\leq 1}$ and the countably supported (sub)distribution monad $\mathcal{D}_{\omega}^{(\leq 1)}$. In the former case, we simply allow distribution to have weight smaller than 1, whereas in the latter case, we allow distributions to have a countable support (i.e. the set of elements where the distribution is non-null must be countable).
4. Computations with output are modelled by the *writer* or *output* monad $(\mathcal{W}, \eta, \mu)$, where $\mathcal{W}A = W \times A$ and $(W, 1, \cdot)$ is a monoid. Unit and multiplication are defined by $\eta(x) = (1, x)$ and $\mu(w, (v, x)) = (w \cdot v, x)$. Taking the monoid of words, then we can think of (w, x) as the result of a program printing w and returning x. If, instead, we take the monoid $(\mathbb{N}, 0, +)$, then we obtain the *cost* or *complexity* monad [59], whereby we read (n, x) as the result of a computation that produces x with cost n. We consider a W-indexed family of unary operations $\mathtt{out}_w$ defined by $\mathtt{out}_w(v, x) = (w \cdot v, x)$. These are indeed algebraic. When dealing with cost, one usually considers the single operation $\mathtt{out}_1$, usually written `tick` or simply $\checkmark$.
5. We model nondeterminism using the powerset monad $(\mathcal{P}, \eta, \mu)$, where $\eta(x) = \{x\}$ and $\mu(\mathcal{U}) = \bigcup \mathcal{U}$. We generate nondeterminism using (binary) set-theoretic union, which is indeed algebraic. The finitary powerset monad $\mathbb{P}_f$ is obtained from $\mathcal{P}$ by taking as underlying functor the finite powerset functor $\mathcal{P}_f$. Similarly, the non-empty powerset monad $\mathcal{P}^+$ is obtained by the taking the non-empty powerset functor $\mathcal{P}^+$.

Most monads seen in Example 2 have *countable support* in the sense that whenever $t \in T(A)$ there exists a countable set $\mathsf{supp}(t) \subseteq A$ upon which t is built. Such a set is called the *support* of t and generalises the notion of a support one has for distributions. In general, not all monads have support and thus we restrict our analysis to such monads. First, we consider only monads that preserve injections: that is, if $\iota : X \hookrightarrow A$ is an injection, then so is $T(\iota) : T(X) \hookrightarrow T(A)$. We regard $T(\iota)$ as a monadic inclusion and write $t \in T(X)$ if there exists (a necessarily unique) $s \in T(X)$ such that $T(\iota)(s) = t$. Notice that all monads of Example 2 preserves injections and that if a monad preserves weak pullbacks (a condition we shall exploit in Section 4), then it preserves injections.

Definition 3 (Support).

1. *Given an element $t \in T(A)$, the support of t, if it exists, is the smallest subset $\iota : X \hookrightarrow A$ such that $t \in T(X)$. We denote such a set by* $\mathsf{supp}(t)$*.*
2. *We say that a monad T has* countable *(resp.* finitary*)* support *if any $t \in T(A)$ has countable (resp. finite) support — i.e.* $\mathsf{supp}(t)$ *exists and it is countable (resp. finite) — for any set A.*

Example 3. All the monads in Example 2 are countable, with the exception of the (non-finitary) powerset monads. Nonetheless, we can regard them as countable by taking their countable restriction. Indeed, as we shall apply them to countable sets (of terms), such a restriction is by no means a constraint. The output monad, the maybe monad, the finitary (sub)distribution monad, and the finitary powerset monad all have finitary support. For example, let us take the set $\mathcal{D}(\mathbb{N})$ of the probability distributions on natural numbers. If $\mathcal{D}(\mathbb{N}) \ni d := \frac{1}{3} \cdot 5, \frac{2}{3} \cdot 7$, applying the definition of support stated above, we obtain that the smallest set X such that $d \in \mathcal{D}(X)$, which is $X = \{5, 7\}$. This set matches exactly the one obtained from the standard definition of support for probability distributions given in Example 2.

Algebraic Theories. Since effects are ultimately produced by algebraic operations, we oftentimes describe computational effects by means of *algebraic theories*, *i.e.* via a collection of operations and equations.

Recall that a *signature* Σ is a family of sets $\{\Sigma_k\}_{k \in \mathbb{N}}$, the elements $\sigma, \rho, \ldots$ of each Σ_k being called k-ary *operations*. The set $T_\Sigma(X)$ of *Σ-terms* (just terms) over X is the least such that (1) $x \in T_\Sigma(X)$ for any $x \in X$, and (2) $\sigma(t_1, \ldots, t_n) \in T_\Sigma(X)$, whenever $t_1, \ldots, t_n \in T_\Sigma(X)$. The construction of Σ-terms defines a functor T_Σ which is part of a monad whose unit is given by the subset inclusion injection $\iota : X \hookrightarrow T_\Sigma X$ and whose multiplication is given by term substitution.

An *algebraic* or *equational theory* over $T_\Sigma(X)$ is given by a relation $E \subseteq T_\Sigma(X) \times T_\Sigma(X)$ of equations between such terms. For a theory E, we write $\sim_E$ (or just $\sim$) for the least congruence relation on terms that is closed under term substitution and contains E. The *free E-theory* over X is the quotient of $T_\Sigma(X)$ by $\sim_E$. This construction gives a functor which is part of a monad, called the *free theory monad* of E.

Example 4 (Concrete Algebraic Theories).

1. The theory of divergence has a single 0-ary operation and no equation. Its free theory monad gives the maybe monad.
2. The theory of nondeterminism consists of a single binary operation $\vee$ together with the usual join semilattice equations [1]. Its associated free theory monad gives the finitary non-empty powerset monad. If we drop the idempotency equation $x \vee x \sim x$, we obtain the theory of multisets [64] and the associated multiset monad. If we also drop commutativity, we obtain the theory of lists and the associated list monad.

3. The theory of probabilistic nondeterminism has binary operations $+_p$ indexed by rational numbers $0 \leq p \leq 1$ subject to the usual axioms of a barycentric algebra [63]:

$$x +_p x \sim x; \qquad x +_1 y \sim x; \qquad x +_p y \sim y +_{1-p} x;$$
$$x +_p (y +_q z) \sim (x +_{\frac{p}{p+(1-p)q}} y) +_{p+(1-p)q} z.$$

   The free theory monad of this theory gives the finitary distribution monad. The theory of multi-distribution or indexed valuations (and their corresponding monads) [7,67,66], is obtained by dropping the idempotency axiom $x +_p x \sim x$.
4. Fixing a monoid W, the theory of the writer monad has a unary operation $\mathtt{out}_w$ for each $w \in W$, and equations

$$\mathtt{out}_1(x) \sim x \qquad \mathtt{out}_w(\mathtt{out}_v(x)) \sim \mathtt{out}_{w \cdot v}(x).$$

Relations. We will extensively work with relations. We denote by **Rel** the category with sets as objects and binary relations as arrows. As it is customary, we use the notation $R : A \nrightarrow B$ for a relation $R \subseteq A \times B$, and write $\mathbf{Rel}(A, B)$ for the collection of relations of type $A \nrightarrow B$. We tacitly regard each function f as a relation via its graph and write $\mathbf{1}_A : A \nrightarrow A$ for the identity relation, the latter being the graph of the identity function. We furthermore denote by $R; S : A \nrightarrow C$ the composition of $R : A \nrightarrow B$ and $S : B \nrightarrow C$, and by $R^\circ : B \nrightarrow A$ the dual or transpose of $R : A \nrightarrow B$.

4 Monadic Intersection Types

In this section, we present the monadic extension of the intersection type system for CbV presented in Section 2.

Effectful CbV. The target calculus of the remaining part of this work is an effectful extension of the CbV λ-calculus previously introduced. We follow the methodology of algebraic effects [57] and fix a signature Σ of effect triggering operations, as seen in the previous section. The calculus $\Lambda_\Sigma^{\mathsf{cbv}}$ is obtained by extending the grammar of the (kernel) CbV λ-calculus as follows:

$$\mathbb{C} \ni t, u ::= \cdots \mid \mathsf{op}(t_1, \ldots, t_n)$$

As before, we denote by $\mathbb{C}^\bullet$ and $\mathbb{V}^\bullet$ the collection of closed computations and values, respectively. Finally, we write $\mathbb{R}^\bullet$ for the subset of $\mathbb{C}^\bullet$ of redexes, *i.e.* computations of the form $(\lambda x.t)v$ or $\mathsf{op}(t_1, \ldots, t_n)$.

We give an operational semantics to $\Lambda_\Sigma^{\mathsf{cbv}}$ in monadic style [56,36]. Let (T, η, μ) be an arbitrary but fixed monad with countable support. We assume that to each n-ary operation $\mathsf{op} \in \Sigma$ it is associated a n-ary algebraic operation g_{op} on T.

We define a function $\mapsto: \mathbb{C}^\bullet \to T(\mathbb{C}^\bullet)$ on closed terms that performs a single computation step (possibly performing effects) by first defining ground reduction and then closing the latter under evaluation contexts. To improve readability, we write $t \mapsto e$ in place of $\mapsto(t) = e$ and we refer to elements in $T(\mathbb{C})$ (resp. $T(\mathbb{V})$) as monadic (or effectful) computations (resp. values).

Definition 4 (Operational Semantics). *We define the function* $\rightarrowtail : \mathbb{R}^\bullet \cup \mathbb{V}^\bullet \to T(\mathbb{C}^\bullet)$ *as follows:*

$$\begin{aligned}(\lambda x.t)v &\rightarrowtail \eta(t\{x/v\})\\ \mathsf{op}(t_1,\ldots,t_n) &\rightarrowtail g_{\mathsf{op}}(\eta(t_1),\ldots,\eta(t_n))\\ v &\rightarrowtail \eta(v)\end{aligned}$$

The function $\mapsto : \mathbb{C}^\bullet \to T(\mathbb{C}^\bullet)$ *is then defined as the contextual closure of* $\rightarrowtail$*, i.e.* $E[r] \mapsto e \gg\!= (\lambda\!\!\!\lambda u.\eta(E[u]))$*, where* r *is a redex and* $r \rightarrowtail e$*.*

In this last definition the symbol $\lambda\!\!\!\lambda$ has to be intended as a meta-lambda notation, *i.e.* by $\lambda\!\!\!\lambda u.\eta(E[u])$ we mean the function $h : \mathbb{C}^\bullet \to T(\mathbb{C}^\bullet)$ such that $h(u) := \eta(E[u])$. In particular, in the definition of the contextual closure we exploit the algebricity of the effects, making them commute with evaluation contexts. Moreover, notice that $\mapsto$ is indeed a function. Consequently, we can rely on its Kleisli extension $\mapsto^\dagger$ to reduce monadic computations. We write $\mapsto^n$ for the n-iteration of $\mapsto$, where $\mapsto^0 := \eta$ and $\mapsto^{n+1} := \mapsto; (\mapsto^n)^\dagger$. In a similar fashion, we write $\mapsto^*$ for $\bigcup_n \mapsto^n$. Please notice that since algebraic operations are finitary, all monadic computations that a computation t can achieve in finite times have finite support, meaning that $t \mapsto^* e$ implies that $\mathsf{supp}(e)$ is finite.

Definition 5 (Finitary Convergence). *We say that a closed computation* t *converges if there exists* $e \in T(\mathbb{C}^\bullet)$ *such that* $t \mapsto^* e$ *and* $\mathsf{supp}(e) \subseteq \mathbb{V}^\bullet$*.*[6] *In that case, we write* $[\![t]\!] = e$*.*

Please notice that $[\![\cdot]\!]$ is a *partial* function. Indeed, terms can diverge.

The Monadic Type System. The main idea behind the development of the type system is that not only terms, but also intersection types become monadic. The natural design choice is to follow the informal CbV translation of intuitionistic logic into linear logic combined with Moggi's translation:

$$A \to B \cong\, !A \multimap T(!B)$$

A third level of (monadic) types is then added to the grammar of types:

$$\begin{array}{rl}\textsc{Value Types} & \mathbb{A} \ni A ::= I \to M\\ \textsc{Intersections} & \mathbb{I} \ni I ::= \{A_1, ..., A_n\}\ n \geq 0\\ \textsc{Monadic Types} & M, N \ni \mathbb{M} ::= T(\mathbb{I})\\ \textsc{Types} & \mathbb{G} \ni G ::= A \mid I \mid M\end{array}$$

[6] To be formally precise, here we should say that $\mathsf{supp}(e)$ belongs to the image of $\mathbb{V}^\bullet$ into $\mathbb{C}^\bullet$. In order to maintain the work as readable as possible, we will be sloppy and here (and in similar situations) simply identify $\mathbb{V}^\bullet$ and its image in $\mathbb{C}^\bullet$.

$$\frac{A \in I}{\Gamma, x : I \vdash x : A}\ \text{VAR} \qquad \frac{[\Gamma \vdash v : I_i \to M_i]_{1 \le i \le n} \quad \Gamma \vdash t : N \quad \mathsf{supp}(N) \subseteq \{I_1, ..., I_n\}}{\Gamma \vdash vt : N \gg\!= (\{I_i \Rightarrow M_i\}_{1 \le i \le n})}\ \text{APP}$$

$$\frac{\Gamma, x : I \vdash t : M}{\Gamma \vdash \lambda x.t : I \to M}\ \text{ABS} \qquad \frac{[\Gamma \vdash t_i : M_i]_{1 \le i \le n}}{\Gamma \vdash \mathsf{op}(t_1, \dots, t_n) : g_{\mathsf{op}}(M_1, \dots, M_n)}\ \text{OP}$$

$$\frac{[\Gamma \vdash v : A_i]_{i \in F}}{\Gamma \vdash v : \{A_i\}_{i \in F}}\ \text{INT} \qquad \frac{\Gamma \vdash v : I}{\Gamma \vdash v : \eta(I)}\ \text{UNIT}$$

Fig. 3: The monadic intersection type system.

We maintain all the notations already presented in Section 2 for the CbV type system. The typing rules are in Fig. 3. While rules ABS, VAR and INT are almost unchanged, the other rules deserve some comments. The rule UNIT is needed to give a monadic type to values. Since values do not produce any effect, they are injected into the monad just with η. Rule APP types applications vt with a monadic type. The important point is that the subterm v in function position has to be typed many times, one for each element in the support of the (monadic) type of the argument t. Please notice (i) that with the notation $\{I_i \Rightarrow M_i\}_{1 \le i \le n}$ we mean the function that maps pointwise I_i to M_i for each $1 \le i \le n$; and (ii) the $\gg\!=$ operator at the level of types. This should not come unexpectedly, since types are monadic, and thus are composed using monadic laws. In particular, the effects produced by the argument, encoded in its type, have to be composed with the effects that will be generated by the rest of the computation (see the example below for more intuitions). Finally, rule OP types operations with the monadic type built with the corresponding algebraic operation, applied to the (monadic) types of the arguments of the operation itself.

Example 5. We provide in Fig. 4 the derivation for $\vdash (\lambda x.x(\mathtt{out}_b(x)))(\mathtt{out}_a(\mathsf{I}))$: $(ab, \mathbf{0})$, which is the very same term as in Example 1, decorated with output operations. As monoid of words, we consider the monoid Σ^* freely generated from the alphabet $\Sigma := \{a, b\}$. One can notice that the assigned type contains all the information about the symbols printed on the output buffer during the evaluation of the term. Please notice how types are composed in the last rule APP. The right-hand side is typed only once, since the support of the monadic type on the left-hand side is a singleton. Notice that the bind operator $\gg\!= : W \times X \to (X \to W \times X) \to W \times X$ in the case of the output monad is defined as: $(a, x) \gg\!= f := (ab, y)$ if $f(x) = (b, y)$. Intuitively, the usual composition is done, but for the fact that the strings (a and b in this case) are concatenated.

Relational Reasoning. To prove soundness and completeness of the monadic type system, we will need to reason both about expressions and monadic expressions. In fact, as long as we are interested in reduction sequences we have to deal both with terms, to start the sequence, and with their (monadic) reducts, to continue the computation. Working with monads, we have already seen that we can extend the dynamic semantic of Λ^{cbv} to monadic computations, for free.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z : \mathbf{0} \vdash z : \mathbf{0}}\;\text{INT}}{z : \mathbf{0} \vdash z : \eta(\mathbf{0})}\;\text{UNIT}}{\vdash \lambda z.z : \texttt{id}}\;\text{ABS}}{\vdash \lambda z.z : \{\texttt{id}\}}\;\text{INT}}{\pi : \vdash \lambda z.z : \eta(\{\texttt{id}\})}\;\text{UNIT}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \{\texttt{id}\} \vdash x : \texttt{id}}\;\text{VAR} \quad \dfrac{\dfrac{\dfrac{}{x : \{\texttt{id}\} \vdash x : \mathbf{0}}\;\text{INT}}{x : \{\texttt{id}\} \vdash x : \eta(\mathbf{0})}\;\text{UNIT}}{x : \{\texttt{id}\} \vdash \texttt{out}_b(x) : (b, \mathbf{0})}\;\text{OP}}{x : \{\texttt{id}\} \vdash x(\texttt{out}_b(x)) : (b, \mathbf{0})}\;\text{APP}}{\vdash \lambda x.x(\texttt{out}_b(x)) : \{\texttt{id}\} \to (b, \mathbf{0})}\;\text{ABS} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : \{\texttt{id}\} \vdash y : \texttt{id}}\;\text{VAR}}{y : \{\texttt{id}\} \vdash y : \{\texttt{id}\}}\;\text{INT}}{y : \{\texttt{id}\} \vdash y : \eta(\{\texttt{id}\})}\;\text{UNIT}}{\vdash \lambda y.y : \{\texttt{id}\} \to \eta(\{\texttt{id}\})}\;\text{ABS} \quad \pi}{\vdash (\lambda y.y)(\lambda z.z) : \eta(\{\texttt{id}\})}\;\text{APP}}{\vdash \texttt{out}_a(\mathsf{II}) : (a, \{\texttt{id}\})}\;\text{OP}}{\vdash (\lambda x.x(\texttt{out}_b(x)))(\texttt{out}_a(\mathsf{II})) : (ab, \mathbf{0})}\;\text{APP}$$

Fig. 4: Type derivation for $\vdash (\lambda x.x(\texttt{out}_b(x)))(\texttt{out}_a(\mathsf{II})) : (ab, \mathbf{0})$. We set $\texttt{id} := \mathbf{0} \to \eta(\mathbf{0}) = \mathbf{0} \to (\varepsilon, \mathbf{0})$.

Computations, however, come both with a dynamic and *static* semantics (*i.e.* types); and if the former semantics (viz. $\mapsto$) extends to monadic computation (as $\mapsto^{\dagger}$), it is not immediately clear how to do the same with the latter. In fact, whereas $\mapsto$ is a function, the static semantics of Λ^{cbv}, given by the typing relation $\vdash$, is genuinely relational, meaning that we cannot rely on the axioms of a monad to extend it.

Relational Extensions. We overcome this problem by relying on the notion of a relational extension of a monad [10,8,11,14,42,39]. Remarkably, relational extensions come with powerful proof techniques, whereby one extends term-based results (such as subject reduction and expansion, in our case) to monadic terms essentially. All of that, however, has a price: relational extensions cannot be given for all monads. As long as we are interested in 'forward properties', such as subject reduction, it is enough to give a relaxed notion of relational extensions — called lax relational extensions — that is available for a large class of monads (such as all the ones seen so far). But if we ask for 'backward properties', such as subject expansion, then we need the full axiomatics of a relational extension. And by a well-known result by Barr [10], we know that a monad has a relational extension if and only if it is *weakly cartesian* [71,15]. From an equational perspective, weakly cartesian monads are defined by affine theories [35], meaning that they cannot have equations that duplicate variables. This excludes important monads, such as the distribution and powerset monads. This way, one has to rely on their 'linearization' to obtain well-behaved intersection type systems. In the case of the distribution and powerset monad one does so simply by dropping the idempotency equations from their equational theories, thus obtaining the so-called multi-distribution [18,66,67,7] and multiset monads [64]. It is important to stress that this is not a design issue, but an intrinsic limit of the model, as shown by Example 8 (below in this section).

Lifting the Type System. The type system in Figure 3 defines a dependent relation $\vdash \in \prod_{\Gamma} \mathcal{P}(\mathbb{A}_{\Gamma} \times \mathbb{G})$, where $\mathbb{A} := \mathbb{V} + \mathbb{C}$ is the collection of terms of the calculus, and $\mathbb{A}_{\Gamma}$ is the collection of terms with free variables among Γ. Notice that $\vdash$ respects syntactic categories, in the sense that since $\mathbb{G} = \mathbb{A} + \mathbb{I} + T(\mathbb{I})$, we can see $\vdash$ as the sum of three relations $\vdash_1 \in \prod_{\Gamma} \mathcal{P}(\mathbb{V}_{\Gamma} \times \mathbb{A})$, $\vdash_2 \in \prod_{\Gamma} \mathcal{P}(\mathbb{V}_{\Gamma} \times \mathbb{I})$, and $\vdash_3 \in \prod_{\Gamma} \mathcal{P}(\mathbb{C}_{\Gamma} \times T(\mathbb{I}))$. In the following, we shall tacitly use this decomposition. Moreover, since type soundness and completeness refer to programs, we will mostly work with $\vdash$ restricted to closed terms (*i.e.* when Γ is empty): in that case, $\vdash$ is just an ordinary binary relation.

When instantiated to monadic types (and closed computations), the relation $\vdash \subseteq \mathbb{C}^{\bullet} \times T(\mathbb{I})$ is a so-called monadic relation [36]. Under suitable conditions on monads, monadic relations come with an operation similar to the Kleisli extension that allows them to be composed and to regard η (seen as a relation) as the unit of such an operation.

Definition 6 (Relational Extension, [10]). *A* relational extension *of a monad* (T, η, μ) *is a family of* monotone *maps* $\Phi : \mathbf{Rel}(A, B) \to \mathbf{Rel}(TA, TB)$ *such that:*

$$\begin{aligned} \mathbf{1} &= \Phi(\mathbf{1}) & \Phi(R);\Phi(S) &= \Phi(R;S) \\ T(f) &= \Phi(f) & \Phi(R)^{\circ} &= \Phi(R^{\circ}) \\ R;\eta &= \eta;\Phi(R) & \Phi(\Phi(R));\mu &= \mu;\Phi(R) \end{aligned}$$

Replacing $=$ *with* $\subseteq$, *we obtain the notion of a* lax relational extension.

Any monad T comes with a canonical candidate relational extension: its Barr extension $\widehat{T}$. Recall that for each relation $R : A \nrightarrow B$, we can regard R as a set $\mathcal{G}(R) \subseteq A \times B$. In particular, the projections $\pi_1 : \mathcal{G}(R) \to A$, $\pi_2 : \mathcal{G}(R) \to B$ give $R = \pi_1^{\circ}; \pi_2$.

Definition 7 (Barr Extension). *The Barr extension* $\widehat{T}$ *of* T *is defined as* $\widehat{T}(R) = (T(\pi_1))^{\circ}; T(\pi_2)$. *Elementwise, we have* $\phi_1 \,\widehat{T}R\, \phi_2$ *iff*

$$\exists \Phi \in T\mathcal{G}(R).\ T(\pi_1)(\Phi) = \phi_1 \text{ and } T(\pi_2)(\Phi) = \phi_2.$$

Example 6 (Concrete Barr Exts.). Let $R : A \nrightarrow B$.

1. For the powerset monad, we have $u \,\widehat{\mathcal{P}}R\, v$ iff $\forall x \in u. \exists y \in v.\ x \mathrel{R} y$ and $\forall y \in v. \exists x \in u.\ x \mathrel{R} y$. A similar definition holds for variations of the powerset monad.
2. For the distribution monad, we have $\phi_1 \,\widehat{\mathcal{D}}R\, \phi_2$ iff there exists $\Phi \in \mathcal{D}(A \times B)$ such that $\sum_y \Phi(x, y) = \phi_1(x)$, $\sum_x \Phi(x, y) = \phi_2(y)$, and $\Phi(x, y) > 0 \implies xRy$. A similar definition holds for variations of the distribution monad.
3. The output monad, we have $(a, x)\ \widehat{\mathcal{W}}(R)\ (b, y)$ iff $a = b$ and $x \mathrel{R} y$.
4. More generally, if a monad is presented by a theory (Σ, E), then we have $t\ \widehat{T_{\Sigma}}(R)\ s$ iff $t \sim_E C[x_1, \ldots, x_n]$, $s \sim_E C[y_1, \ldots, y_n]$, and $x_i \mathrel{R} y_i$, for any i.

The Barr extension of a monad is not a relational extension, in general. However, the Barr extension of a monad is a relational extension iff the monad is weakly cartesian [71,15].

Theorem 2 ([10]). *Recall that a monad (T, η, μ) is* weakly cartesian *if (i) it preserves weak pullbacks and (ii) all naturality squares of η and μ are weak pullbacks. If T is weakly cartesian, then its Barr extension is the* unique *relational extension of T. If T preserves weak pullbacks, then its Barr extension is a lax relational extension.*

For brevity, we say that a monad is WC—resp. WP—if it is weakly cartesian—if it preserves weak pullbacks.

Example 7. All the monads seen so far are WP. The output and maybe monad, additionally, are WC, whereas the powerset and distribution monads are not [71], as naturality squares of their unit are not weak pullbacks. If a monad T is presented by an affine equational theory [35] (Σ, E), meaning that all equations in E are affine, then it is WC. Consequently, the multiset and multidistribution monad are WC.

Given a monad T and a monadic relation $R \subseteq A \times T(B)$, we define its Kleisli extension $R^\dagger \subseteq T(A) \times T(B)$ as $\widehat{T}(R); \mu$. Using Kleisli extension, we define the composition of monadic relations $R \subseteq A \times T(B)$ and $S \subseteq B \times T(C)$ as the relation $R \mathbin{⨟} S \subseteq A \times T(C)$ defined as $R; S^\dagger$. If T is WC (resp. WP), $⨟$ is (lax) associative and has η as (lax) unit [36,40]. Using the Kleisli extension we can design abstract proof techniques ensuring that properties of $\vdash$ with respect to the one-step semantics $\mapsto$ can be lifted to $\vdash^\dagger$ and $\mapsto^\dagger$.

Proposition 3 ([10]). *Let $R : A \nrightarrow B$ be a relation and Φ be a lax relational extension of a monad T. Then, (i) $\Phi(R)$ is closed under algebraic operation; (ii) $\Phi(R)$ is closed under monadic binding: $R; g \subseteq f; \Phi(S)$ implies $\Phi(R); g^\dagger \subseteq f^\dagger; \Phi(S)$.*

The next result will be crucial to prove subject expansion.

Proposition 4. *Let T be WC. Then: $f; R^\dagger \subseteq S \implies f^\dagger; R^\dagger \subseteq S^\dagger$.*

In particular, taking both R and S as the typability relation $\vdash \subseteq \mathbb{C} \times T(\mathbb{I})$ and as f the one-step semantic function $\mapsto : \mathbb{C} \to T(\mathbb{C})$, we see that $\mapsto; \vdash^\dagger \subseteq \vdash$ states that whenever we have a term t with $t \mapsto e$ and a monadic type M with $\vdash^\dagger e : M$, then $\vdash t : M$. This is exactly the statement of the subject expansion theorem at the level of term-based evaluation that we shall prove in the next section: if $t \mapsto e$ and $\vdash^\dagger e : M$, then $\vdash t : M$. Prop. 4 then implies that subject expansion can be extended to full monadic reduction $\mapsto^\dagger$: if $e \mapsto^\dagger e'$ and $\vdash^\dagger e' : M$, then $\vdash^\dagger e : M$.

Obviously, Proposition 4 still requires us to prove $\mapsto; \vdash^\dagger \subseteq \vdash$, and we would like to do so syntactically. Although natural, this relational extension is not always possible. The problem lies in the fact that if we assign a monadic type M to an element of the form $\eta(t)$ via $\vdash^\dagger$, there is no guarantee that t itself has type M. This becomes problematic when dealing with values. Since a value v (regarded as a computation) reduces to $\eta(v)$ and our monadic type system assigns only types of the form $\eta(I)$ to v, provided that $\vdash v : I$, we need to ensure that any type M such that $\vdash^{\dagger}\eta(v) : M$ is itself a type of v, and hence of the form $\eta(I)$. This, however, is not always the case.

Example 8. Let us consider the distribution monad $\mathcal{D}$ and recall that its unit maps a point to its Dirac distribution. Let v be a value such that $\vdash v : A$ and $\vdash v : B$, so that $\vdash v : \{A\}$ and $\vdash v : \{B\}$. By the very definition of the monadic type system, the computation induced by the value v can only have monadic types of the form $\eta(I)$ (*i.e.* Dirac distributions $1 \cdot I$). Yet, the lifted relation $\vdash^{\dagger}$ gives $\vdash^{\dagger}\eta(v) : \frac{1}{2} \cdot \{A\} + \frac{1}{2} \cdot \{B\}$, since $\eta(v) = 1 \cdot v = \frac{1}{2} \cdot v + \frac{1}{2} \cdot v$ and $\vdash v : \{A\}$ and $\vdash v : \{B\}$ entail $\vdash^{\dagger} 1 \cdot v : 1 \cdot \{A\}$ and $\vdash^{\dagger} 1 \cdot v : 1 \cdot \{B\}$. Consequently, we have $\vdash^{\dagger} \eta(v) : \frac{1}{2} \cdot \{A\} + \frac{1}{2} \cdot \{B\}$ but we cannot have $\vdash v : \frac{1}{2} \cdot \{A\} + \frac{1}{2} \cdot \{B\}$.

The ultimate source of the problem outlined in the above example is that the unit of $\mathcal{D}$ is not weakly cartesian.

Proposition 5. *Let T be WC. For any monadic relation $R \subseteq A \times T(B)$, we have $\eta; R^{\dagger} = R$. In particular, $\eta; \vdash^{\dagger} = \vdash$.*

The techniques seen so far have been designed to prove subject expansion of the monadic type system. As expected, we are also interested in proving subject reduction and thus it is natural to design similar proof techniques in that setting. This can be easily done following the same path as for subject expansion, but with a main difference: subject reduction does not require the unit of the monad to be WC, and hence subject reduction results can be proved for a much larger class of monads.[7]

Proposition 6. *Let T be WP. Then: $R^{\circ}; f \subseteq R^{\dagger\circ} \implies R^{\dagger\circ}; f^{\dagger} \subseteq R^{\dagger\circ}$.*

In particular, we can instantiate Proposition 6 with R and f as $\vdash$ and $\mapsto$, hence obtaining $\vdash^{\circ}; \mapsto \subseteq \vdash^{\dagger\circ} \implies \vdash^{\dagger\circ}; \mapsto^{\dagger} \subseteq \vdash^{\dagger\circ}$, meaning that whenever subject reduction holds at the level of terms (*i.e.* $\vdash t : M \ \& \ t \mapsto e \implies \vdash^{\dagger} e : M$), then it holds at the level of their (monadic) evaluation (*i.e.* $\vdash^{\dagger} e : M$ and $e \mapsto^{\dagger} e'$ implies $\vdash^{\dagger} e' : M$).

Soundness. The proof of soundness of the type system consists in showing:
1. *Subject reduction:* types are preserved by reduction.
2. *Termination:* all typable terms terminate.

As we have seen, by Proposition 6, it is sufficient to prove subject reduction with respect to the single-step, term-based reduction $\mapsto$. This is done by induction on the structure of evaluation contexts, with the help of a substitution lemma, proved by induction on the structure of terms.

Proposition 7 (Subject Reduction). *Let T be WP. Then:*
1. *Let t be a closed λ-term. If $\vdash t : M$ and $t \mapsto e$, then $\vdash^{\dagger} e : M$.*
2. *Let e be a monadic closed λ-term. If $\vdash^{\dagger} e : M$ and $e \mapsto^{\dagger} e'$, then $\vdash^{\dagger} e' : M$.*

Proving termination instead needs more work.

[7] Notice that even if η is not WC, we still have $R; \eta \subseteq \eta; \Phi R$ (but not the other inclusion, which is crucial in Proposition 5).

Effectful Observations. Knowing that typing is preserved by reduction, it remains to show that whenever a computation t has type M, its observable operational behaviour is fully captured by M. In the pure case, such a behaviour is just termination, so that one usually shows that typable terms terminate. In the effectful setting, termination can be given in many forms. First, if effects capture some forms of nondeterminism, meaning that elements in $T\mathbb{A}$ may have more than one element in their support, then termination can be divided into *may* or *must* termination (*i.e.* whether term reaches monadic expressions with one, at least, or all values in their support). In both of these cases, termination remains a boolean notion (viz. a predicate). To account for effects it is natural to ask not only whether a computation terminates, but also which effects are produced during evaluation (*e.g.* what is stored in memory locations, which are the printed outputs, the cost of the computation, etc). A further option is to make termination effectful itself, a well-known example of effectful termination being almost-sure termination (*i.e.* probability of convergence). Such notions are usually infinitary and require non-boolean reasoning.

Since here we deal with the finitary case, we agree to observe must termination of computations as well as the effects produced during their evaluation. In the next section, we shall deal with infinitary evaluation and, consequently, with effectful termination. Let us begin by formalising how to observe effects. In a monadic setting, it is customary [19,20,21,62], to model (effectful) observables as elements of $T(X)$, where X is what is observable of expressions. As we are interested in must termination, only values are observable, and, moreover, they cannot be scrutinized further (*i.e.* we observe that a computation gives a result (a value), but we cannot inspect such a result[8]).

Definition 8. *We define the observation function for monadic objects in* $T(X)$ *as* $\mathsf{obs}_X := T(!_X) : T(X) \to T(1)$*, where* $1 := \{\star\}$ *and* $!_X : X \to 1$ *is the unique arrow collapsing all the elements of* X *to* $\star$*. We extend* $\mathsf{obs}_{\mathbb{A}}$ *to a partial function on terms by stipulating* $\mathsf{obs}_{\mathbb{A}}(t) := \mathsf{obs}_{\mathbb{A}}(e)$*, provided that* $[\![t]\!] = e$*.*

As usual, we omit subscripts whenever possible, writing $\mathsf{obs}(e)$, $\mathsf{obs}(M)$, etc.

Example 9 (Concrete Observations).

1. The output, or writer, monad $\mathcal{W}$ has a notion of observation $\mathsf{obs} : \mathcal{W}(X) \to W$, if W is the underlying monoid of words. This is immediate to see because $\mathcal{W}(1) := W \times \{\star\} \cong W$. Then, we have that $\mathsf{obs}((w, x)) := w$. This means that what we can observe is the string that has been printed on the output buffer during the computation.
2. The partiality monad $\mathcal{E}$ provides a binary notion of observation, indeed $\mathsf{obs} : \mathcal{E}(X) \to \{\star, \bot\}$. This is actually the way in which one could observe divergence.
3. The powerset monad $\mathcal{P}$ comes with the natural notion of must termination, since $\mathsf{obs} : \mathcal{P}(X) \to \{\emptyset, 1\}$.

[8] This is standard in weak, untyped λ-calculi. One could add constants, such as booleans, or numerals, and then observe their shape, in a straightforward way.

Remark 2. According to Definition 8, the observable effects produced by a computation are elements of $T(1)$. This certainly works well for some effects and monads, such as output and cost, but it may be unusual for others. For instance, probabilistic nondeterminism is usually modelled using (variations of) the sub-distribution monad $\mathcal{D}$ and, since $\mathcal{D}(1) \cong [0,1]$, it is natural to interpret elements in the latter set as actual probabilities of events (such as the probability of termination, in our case). However, we have already seen that it is simply not possible to have well-behaved forms of intersection types working with $\mathcal{D}$, and that we can overcome that issue by working with the multi-sub-distribution monad $\mathcal{M}$. Unfortunately, $\mathcal{M}(1) \ncong [0,1]$, although we would still like to think about the observable behaviour of a program as its probability of convergence. This is not a big issue since it does not take much to realise that our analysis of observations works *mutatis mutandis* if we replace $T(1)$ with $S(1)$, where S is another monad such that there is a monad morphism $\nu : T \Rightarrow S$. This way, for instance, even if modelling static and dynamic semantics in terms of $\mathcal{M}$, we can regard $\mathsf{obs}(t)$ as the probability convergence of convergence of t, due to the monad morphism $\nu : \mathcal{M} \Rightarrow \mathcal{D}$ collapsing multi-sub-distributions into ordinary sub-distributions.

Termination by Logical Relations. Our goal is now to prove that whenever a term t has type M then: (i) t must terminate, and (ii) the observable behaviour of t, *i.e.*, $\mathsf{obs}(t)$, is fully described by M. That is, $\mathsf{obs}(t) = \mathsf{obs}(M)$. To achieve such a goal, we define a logical relation $\models$ between (closed) terms and types acting as the semantic interpretation of $\vdash$ in such a way that $\vdash \, = \, \models$ and $\models t : M$ implies $t \Downarrow e$ (must termination) and $\mathsf{obs}(e) = \mathsf{obs}(M)$. Remarkably, such a logical relation makes crucial use of the Barr extension of T.

Definition 9. *We define the logical semantics of* $\vdash$ *(restricted to closed expressions) as the relation* $\models \, := \, \models_{\mathbb{A}} + \models_{\mathbb{I}} + \models_{\mathbb{M}}$ *that inductively refines* $\vdash$ *(i.e.* $\models \, \subseteq \, \vdash$*) as follows, where we use the notation* $\widehat{\models} e : M$ *in place of* $e \; \widehat{T}(\models_{\mathbb{I}}) \; M$*.*

$$\begin{array}{lll} \models_{\mathbb{A}} v : I \to M & \textit{iff} & \forall w.\ \models_{\mathbb{I}} w : I \textit{ implies } \models_{\mathbb{M}} vw : M \\ \models_{\mathbb{I}} v : \{A_1, .., A_n\} & \textit{iff} & \forall i.\ \models_{\mathbb{A}} v : A_i \\ \models_{\mathbb{M}} t : M & \textit{iff} & \widehat{\models_{\mathbb{I}}} \llbracket t \rrbracket : M \end{array}$$

As usual, we omit subscripts whenever unambiguous. We first show that, indeed, $\models$ ensures the desired property.

Lemma 3. $\models t : M$ *implies* $\exists e$ *such that* $\llbracket t \rrbracket = e$ *and* $\mathsf{obs}(e) = \mathsf{obs}(M)$*.*

Then, we prove the soundness of our type system showing that $\models$ and $\vdash$ coincide.

Proposition 8 (Soundness). $\vdash \, = \, \models$.

Completeness. Having proved soundness of our type system, we now move on to completeness, meaning that normalising terms are typable. The proof of completeness follows the usual pattern for intersection types, and makes crucial use of subject expansion. The proof of subject expansion is divided into two parts: first, we prove subject expansion with respect to the single-step reduction on terms and then extend such a result to monadic terms and monadic reduction relying on Proposition 4. Concerning the first part, we would like to prove subject expansion by induction on the structure of evaluation contexts (after having proved a straightforward anti-substitution lemma). However, the statement is not true in general.

Example 10. Let us consider the multidistribution monad and the binary operation $\oplus_1$, *i.e.* the first projection. Let us consider the reduction $\lambda x.x \oplus_1 \Omega \mapsto 1 \cdot \lambda x.x$. Even if $\vdash^{\dagger} 1 \cdot \lambda x.x : 1 \cdot M$, it is not possible to type $\lambda x.x \oplus_1 \Omega$ with rule OP, because of course there is no way of typing Ω.

Then, we need a restriction on our calculus, this time about operations. We allow only operations $\mathsf{op}(t_1, \ldots, t_n)$ that do *not* erase their arguments, *i.e.* for which $\mathsf{supp}(g_{\mathsf{op}}(\eta(t_1), \ldots, \eta(t_n))) = \{t_1, \ldots, t_n\}$. In terms of equational theories, this is guaranteed by considering *linear* theories. We already anticipate that we will be able to remove this restriction in the next section, by the use of infinitary means.

Proposition 9 (Subject Expansion). *Let T be WC. Then:*

1. *If $t \mapsto e$ and $\vdash^{\dagger} e : M$, then $\vdash t : M$.*
2. *If $e \mapsto^{\dagger} e'$ and $\vdash^{\dagger} e' : M$, then $\vdash^{\dagger} e : M$.*

Proposition 9, together with the fact that monadic values can always be typed, gives the completeness of the type system.

Theorem 3 (Completeness). *If $[\![t]\!] = e$, then there exists a monadic type M such that $\vdash t : M$.*

Soundness and completeness together provide a characterization of finitary effectful termination via typability with intersection types.

Corollary 1 (Characterization). *The following clauses are equivalent:*

1. *Effectful termination:* $\mathsf{obs}([\![t]\!]) = o$.
2. *Typability: there exists M, such that $\vdash t : M$ and* $\mathsf{obs}(M) = o$.

5 Infinitary Effectful Semantics

In this section, we extend the type system of Section 4 to account for infinitary behaviours. To do so, we require monads to have enough structure to support such behaviours. A standard approach to do that is by requiring suitable order-theoretic enrichments. Here, we consider monads whose Kleisli category is enriched in the category of directed complete pointed partial order (dcppos) [43,1], but in order to maintain the paper as self-contained as possible, we use the following more concrete (and restricted) definition.

$$\Phi : \quad \dfrac{\dfrac{\dfrac{}{x : \{\mathtt{id}\} \vdash x : \mathtt{id}}\ \textsc{var} \qquad \dfrac{\dfrac{}{x : \{\mathtt{id}\} \vdash x : \mathbf{0}}\ \textsc{int}}{x : \{\mathtt{id}\} \vdash x : \eta(\mathbf{0})}\ \textsc{unit}}{x : \{\mathtt{id}\} \vdash xx : \eta(\mathbf{0})}\ \textsc{app}}{\vdash \lambda x.xx : \{\mathtt{id}\} \to \eta(\mathbf{0})}\ \textsc{abs}$$

$$\Psi : \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : \{\mathtt{id}\} \vdash y : \mathtt{id}}\ \textsc{var}}{y : \{\mathtt{id}\} \vdash y : \{\mathtt{id}\}}\ \textsc{int}}{y : \{\mathtt{id}\} \vdash y : 1\{\mathtt{id}\}}\ \textsc{unit}}{\vdash \lambda y.y : \{\mathtt{id}\} \to 1\{\mathtt{id}\}}\ \textsc{abs} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z : \mathbf{0} \vdash z : \mathbf{0}}\ \textsc{int}}{z : \mathbf{0} \vdash z : \eta(\mathbf{0})}\ \textsc{unit}}{\vdash \lambda z.z : \mathtt{id}}\ \textsc{abs}}{\vdash \lambda z.z : \{\mathtt{id}\}}\ \textsc{int}}{\vdash \lambda z.z : 1\{\mathtt{id}\}}\ \textsc{unit}}{\vdash (\lambda y.y)(\lambda z.z) : 1\{\mathtt{id}\}}\ \textsc{app} \qquad \dfrac{}{\vdash \Omega : \bot}\ \textsc{bot}}{\vdash \mathsf{I} \oplus \Omega : \frac{1}{2}\{\mathtt{id}\}}\ \textsc{op}$$

$$\dfrac{\Phi \qquad \Psi}{\vdash (\lambda x.xx)(\mathsf{I} \oplus \Omega) : \frac{1}{2}\mathbf{0}}\ \textsc{app}$$

Fig. 5: Type derivation for $\vdash (\lambda x.xx)(\mathsf{I} \oplus \Omega) : \frac{1}{2}\mathbf{0}$. $\mathtt{id} := \mathbf{0} \to \eta(\mathbf{0}) = \mathbf{0} \to 1{\cdot}\mathbf{0}$.

Definition 10. *A monad $(T, \eta, \gg\!=)$ is dcppo-ordered if, for any set A, we have a dcppo $(T(A), \sqsubseteq_A, \bot_A)$ such that the bind operator is strict and continuous in both arguments.*

As usual, we omit subscripts whenever unambiguous. Notice that if T is dcppo-ordered, then all its algebraic operations are strict and continuous.

Example 11. Both the multiset and multidistribution monad can be turned into dcppo-ordered monads by simply adding a zero-ary operation symbol $\bot$ to their equational theories. Semantically, $\bot$ corresponds to the empty multiset and multidistribution, respectively.

From now on, we tacitly work with an arbitrary but fixed dcppo-ordered monad $(T, \eta, \gg\!=)$.

Infinitary Typing. Extending the monadic type system to the infinitary case is straightforward. We simply add the typing rule

$$\dfrac{}{\Gamma \vdash t : \bot}\ \textsc{bot}$$

allowing to type any computation with the total uninformative type $\bot$. Consequently, we can assign several types to each term.

Example 12. We provide in Fig. 5 the type derivation for the term $\vdash (\lambda x.xx)(\mathsf{I} \oplus \Omega) : \frac{1}{2}\mathbf{0}$, again a simple variation on the theme of the previous examples. One can notice that we are able to type it, even if clearly the term does *not* converge. Its type $\frac{1}{2}\mathbf{0}$ says exactly that: the probability of convergence is $\mathsf{obs}(\frac{1}{2}\mathbf{0}) = \frac{1}{2}$.

Nonetheless, we can think about type derivations (with occurrences of the BOT rule) as approximations of the semantic content of a computation, the latter being reached only at the limit. Moreover, the set of the observations $\mathcal{O}(t)$ of a term t, defined as the collection of all the observations $\mathsf{obs}(M)$ for $\vdash t : M$ is directed. Consequently, we can associate to each t a more informative observation obtained through types given as $O(t) := \bigsqcup \mathcal{O}(t)$. Notice that even if $O(t)$ is a valid observation, there may be no (necessarily finite) derivation $\pi \triangleright \vdash t : M$ such that $\mathsf{obs}(M) = O(t)$.

Infinitary Operational Semantics. A standard approach to deal with infinitary effectful semantics consists in defining a monadic evaluation function mapping computations to monadic values. To capture forms of convergence in the limit, such a function is defined as a suitable least upper bound of maps evaluating computations for a fixed number of steps. We implement this strategy building upon the definition of $\mapsto$.

Definition 11 (Approximate Operational Semantics). *Let $\phi : \mathbb{C} \to T(\mathbb{V})$ mapping values v (as elements in $\mathbb{C}$) to $\eta(v)$ and all other terms t to $\bot$. Then, we define the $\mathbb{N}$-indexed family of maps $[\![-]\!]^n : \mathbb{C}^\bullet \to T(\mathbb{V}^\bullet)$ by $[\![t]\!]^0 := \bot$, and $[\![t]\!]^n := e \gg\!= \phi$, if $n > 0$ and $t \mapsto^n e$.*

Lemma 4 ([19]). *For any closed computation t, the sequence $\{[\![t]\!]^n\}_{n \geq 0}$ forms a directed set (an ω-chain, actually).*

Consequently, we define (overriding the previous finitary definition) $[\![t]\!] = \bigsqcup_n [\![t]\!]^n$. Notice that this also gives a straightforward way to extend the observation function obs (on terms) to the infinitary setting. We simply define $\mathsf{obs}(t) := T(!)([\![t]\!])$, with $! : \mathbb{V} \to 1$ be as before. Moreover, since T is dcppo-enriched, obs is continuous (and thus monotone). In particular, we can define a bounded observation function as $\mathsf{obs}^n(t) := T(!)([\![t]\!]^n)$ and see that $\mathsf{obs}(t) = \bigsqcup_n \mathsf{obs}^n(t)$. We now have all the ingredients needed to extend our characterization to the infinitary setting.

6 Characterizing Infinitary Behaviors

In this section, we extend the soundness and completeness results previously seen to the infinitary setting. Remarkably, most of the proofs given in the finitary case, such as those of subject reduction and expansion, scale to the infinitary case. This is no coincidence but a main strength of the abstract relational approach that we have developed in the previous part of this work.

Soundness. As in the finitary case, we have subject reduction for WP monads.

Proposition 10 (Subject Reduction, Infinitary). *Let T be WP. Then:*

1. *Let t be a closed λ-term. If $\vdash t : M$ and $t \mapsto e$, then $\vdash^\dagger e : M$.*
2. *Let e be a monadic closed λ-term. If $\vdash^\dagger e : M$ and $e \mapsto^\dagger e'$, then $\vdash^\dagger e' : M$.*

Notice that Proposition 10 is given relying on the Barr extension of T which, by its very definition, does not take into account the order $\sqsubseteq$ induced by T. In particular, whenever we have $\vdash^\dagger e : M$, then e and M must have the same effectful behaviour. This means that as long as we stick with $\widehat{T}(\vdash)$, it is simply not possible to extend subject reduction (and thus soundness) to the full evaluation $[\![-]\!]$, the latter being infinitary. Consequently, contrary to the finitary case, there is no hope to prove that whenever $\vdash t : M$, then M encodes the whole observable behaviour of t, i.e. $\mathsf{obs}([\![t]\!])$. What we can show, however, is that M provides an approximation of such a behaviour, and that the limit of such approximations is precisely the operational behaviour of t. The right tool to achieve such a goal, is an *ordered* version of the Barr extension [41].

Definition 12 (Right Barr Extension). *Given $R \subseteq A \times B$, we define its* right Barr extension $\widehat{T}_{\sqsupseteq}(R) \subseteq T(A) \times T(B)$ *as* $\widehat{T}_{\sqsupseteq}(R) := \widehat{T}(R); \sqsupseteq$.

Proposition 11 ([41]). *If T is WP, then $\widehat{T}_{\sqsupseteq}$ is a lax relational extension.*

Using the right Barr extension, we define the logical relation interpreting $\vdash$.

Definition 13 (Infinitary Logical Relation). *We define the logical semantics of $\vdash$ (restricted to closed expressions) as the relation $\models := \models_{\mathbb{A}} + \models_{\mathbb{I}} + \models_{\mathbb{M}}$ that inductively refines $\vdash$ (i.e. $\models \subseteq \vdash$) as follows, where we use the notation $\widetilde{\models}\ e : M$ in place of $e\ \widehat{T}_{\sqsupseteq}(\models)\ M$.*

$$\begin{array}{ll}
\models_{\mathbb{A}} v : I \to M & \textit{iff } \forall w.\ \models_{\mathbb{I}} w : I \textit{ implies } \models_{\mathbb{M}} vw : M \\
\models_{\mathbb{I}} v : \{A_1, ..., A_n\} & \textit{iff } \forall i.\ \models_{\mathbb{A}} v : A_i \\
\models_{\mathbb{M}} t : M & \textit{iff } \widetilde{\models}\ [\![t]\!] : M
\end{array}$$

As usual, we omit subscripts whenever unambiguous.

Lemma 5. $\models t : M$ *implies* $\mathsf{obs}([\![t]\!]) \sqsupseteq \mathsf{obs}(M)$.

As in the finitary case, we prove the soundness of our type system showing that $\models$ and $\vdash$ coincide.

Proposition 12 (Soundness). *If T is WP, then $\vdash = \models$.*

Completeness. As in the finitary case, completeness is proved via subject expansion. This latter result, in turn, is obtained exactly as in the finitary case. The only difference is that we are able to drop the constraint about non-erasing operations. Indeed, this time we can type erased (and thus possibly diverging) arguments with the rule BOT.

Proposition 13 (Subject Expansion, Infinitary). *Let T be WC. Then:*

1. *If $t \mapsto e$ and $\vdash^\dagger e : M$, then $\vdash t : M$.*
2. *If $e \mapsto^\dagger e'$ and $\vdash^\dagger e' : M$, then $\vdash^\dagger e : M$.*

$$\Phi : \quad \dfrac{\dfrac{\dfrac{\dfrac{}{x:\{\mathtt{id}\}\vdash x:\mathtt{id}}\,\text{VAR} \qquad \dfrac{\dfrac{}{x:\{\mathtt{id}\}\vdash x:\mathbf{0}}\,\text{INT}}{x:\{\mathtt{id}\}\vdash x:\eta(\mathbf{0})}\,\text{UNIT}}{x:\{\mathtt{id}\}\vdash xx:\eta(\mathbf{0})}\,\text{APP}}{x:\{\mathtt{id}\}\vdash \checkmark(xx):1{\cdot}(1,\mathbf{0})}\,\text{OP}}{\vdash \lambda x.\checkmark(xx):\{\mathtt{id}\}\to 1{\cdot}(1,\mathbf{0})}\,\text{ABS}$$

$$\Psi : \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y:\{\mathtt{id}\}\vdash y:\mathtt{id}}\,\text{VAR}}{y:\{\mathtt{id}\}\vdash y:\{\mathtt{id}\}}\,\text{INT}}{y:\{\mathtt{id}\}\vdash y:1{\cdot}(0,\{\mathtt{id}\})}\,\text{UNIT}}{\vdash \lambda y.y:\{\mathtt{id}\}\to 1{\cdot}(0,\{\mathtt{id}\})}\,\text{ABS} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z:\mathbf{0}\vdash z:\mathbf{0}}\,\text{INT}}{z:\mathbf{0}\vdash z:\eta(\mathbf{0})}\,\text{UNIT}}{\vdash \lambda z.z:\mathtt{id}}\,\text{ABS}}{\vdash \lambda z.z:\{\mathtt{id}\}}\,\text{INT}}{\vdash \lambda z.z:1{\cdot}(0,\{\mathtt{id}\})}\,\text{UNIT}}{\vdash (\lambda y.y)(\lambda z.z):1{\cdot}(0,\{\mathtt{id}\})}\,\text{APP}}{\vdash \checkmark(\mathsf{II}):1{\cdot}(1,\{\mathtt{id}\})}\,\text{OP} \qquad \begin{matrix}\vdots\\ \vdash \mathsf{I}:1{\cdot}(0,\{\mathtt{id}\})\end{matrix}}{\vdash \checkmark(\mathsf{II})\oplus\mathsf{I}:\frac{1}{2}{\cdot}(1,\{\mathtt{id}\}),\frac{1}{2}{\cdot}(0,\{\mathtt{id}\})}\,\text{OP}$$

$$\dfrac{\dfrac{\Phi \qquad \Psi}{\vdash (\lambda x.\checkmark(xx))(\checkmark(\mathsf{II})\oplus\mathsf{I}):\frac{1}{2}{\cdot}(2,\mathbf{0}),\frac{1}{2}{\cdot}(1,\mathbf{0})}\,\text{APP}}{\vdash \checkmark((\lambda x.\checkmark(xx))(\checkmark(\mathsf{II})\oplus\mathsf{I})):\frac{1}{2}{\cdot}(3,\mathbf{0}),\frac{1}{2}{\cdot}(2,\mathbf{0})}\,\text{OP}$$

Fig. 6: Type derivation for $\vdash \checkmark((\lambda x.\checkmark(xx))(\checkmark(\mathsf{II})\oplus\mathsf{I})):\frac{1}{2}{\cdot}(3,\mathbf{0}),\frac{1}{2}{\cdot}(2,\mathbf{0})$. We set $\mathtt{id}:=\mathbf{0}\to\eta(\mathbf{0})=\mathbf{0}\to 1{\cdot}(0,\mathbf{0})$.

Then, we are able to prove approximate completeness, by finitary means.

Theorem 4 (Approximate Completeness). *Let t be a closed λ-term. Then, for each $k\geq 0$, there exist $\pi_k \triangleright \vdash t : M_k$ such that $\mathsf{obs}(M_k)=\mathsf{obs}^k(t)$.*

Finally, we can claim the full characterization of the infinitary effectful behavior of any program by the way of our intersection type system.

Corollary 2 (Characterization). *Let t be a closed λ-term. Then $O(t)=\mathsf{obs}(t)$.*

Example 13. As a concluding example, in Fig. 6 we show the type derivation for the term $\vdash \checkmark((\lambda x.\checkmark(xx))(\checkmark(\mathsf{II})\oplus\mathsf{I})):\frac{1}{2}{\cdot}(3,\mathbf{0}),\frac{1}{2}{\cdot}(2,\mathbf{0})$. Please notice that this example is built on the composition of two different monads: cost and multidistribution. This way, we show how we are able to handle computational effects in a modular way. It is easy to verify that the same would have been possible, *e.g.*, for cost and multipowerset. $\mathsf{obs}(\frac{1}{2}{\cdot}(3,\mathbf{0}),\frac{1}{2}{\cdot}(2,\mathbf{0}))=\frac{1}{2}{\cdot}3,\frac{1}{2}{\cdot}2$, which is a *distribution* on *costs*. Taking its expected value, one can indeed obtain the average cost of the computation. One can build an example that actually uses infinitely many types (and derivations) on the same line of the one presented in [18].

7 Conclusion

In this paper, we have proposed the first intersection type system able to characterize the effectful behavior for terms of the λ-calculus enriched with algebraic operations. In particular, we are able to do that parametrically with respect to the underlying monad. Moreover, having presented effects as algebraic theories, it is possible to compose effects relying both on the sum and tensor of algebraic theories. Since effectful behaviors are often observed at the limit, we had to deal with infinitary constructions. Technically speaking, relational reasoning was the main tool exploited to obtain our result in an abstract and modular way.

Perspectives. This work opens several research directions:

- *Quantitative Cost Analyses:* Idempotent intersection types are qualitative in nature because they are not able to track the use of resources, such as time or space, during the evaluation. Turning intersections (*i.e.* sets) into multisets is enough to measure the precise cost of the evaluation of typed terms, while maintaining the correctness of the type system [24,4,3]. Extending this machinery to the effectful setting would be very interesting, although not trivial. While there are standard notions of monadic costs (*e.g.* the *average* cost in the probabilistic setting, or the *maximum* cost in must nondeterminism), it is not clear how to devise the type system to capture them. In the probabilistic case, for example, some additional information had to be stored inside types to correctly compute the average number of steps [18]. Very recently some investigations on the state and the exception monad (featuring also handling) have appeared [5,44], but the design of the type systems seems ad-hoc and not easy to generalize.
- *Higher-Order Model Checking:* Model checking of higher-order recursion schemes has been proven decidable by Ong in 2006 [52]. Since then, several papers dissected the original result and gave other proof methods and model checking algorithms. Among them, Kobayashi and coauthors developed type-theoretic techniques based on intersection types [48,46]. While the literature contains results about model checking higher-order programs enriched with specific effects, such as probability [47] or nondeterminism [65], no general method covering families of computational effects is known. Indeed, we would like to investigate if our type system could guide the synthesis of model checking algorithms in the style of [65]. Since the problem has been proved in general undecidable in the effectful setting, *e.g.* in the case of the sub-distribution monad [47], one would need of course to restrict the class of monads in order to recover decidability.
- *Adding Coinduction:* Our type system is not able to deal with coinductive properties, such as productivity, or with coinductive effects, like the output of streams. We would like to enhance the type system with coinductive types/rules in order to capture these kind of properties. Since the type system is somehow modelled on top of operational semantics, this would require to change it as well. We mention that very recently some works covering coinduction have appeared, but limited to the pure λ-calculus, and carried on in the non-idempotent setting [68,69].

Acknowledgments. To Ugo Dal Lago, who set the basis for this work.

References

1. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science. pp. 1–168. Clarendon Press (1994)
2. Accattoli, B.: Proof nets and the call-by-value λ-calculus. Theor. Comput. Sci. **606**, 2–24 (2015). https://doi.org/10.1016/j.tcs.2015.08.006, https://doi.org/10.1016/j.tcs.2015.08.006
3. Accattoli, B., Dal Lago, U., Vanoni, G.: Multi types and reasonable space. Proc. ACM Program. Lang. **6**(ICFP), 799–825 (2022). https://doi.org/10.1145/3547650
4. Accattoli, B., Graham-Lengrand, S., Kesner, D.: Tight typings and split bounds, fully developed. J. Funct. Program. **30**, e14 (2020). https://doi.org/10.1017/S095679682000012X
5. Alves, S., Kesner, D., Ramos, M.: Quantitative global memory. In: Hansen, H.H., Scedrov, A., de Queiroz, R.J.G.B. (eds.) Logic, Language, Information, and Computation - 29th International Workshop, WoLLIC 2023, Halifax, NS, Canada, July 11-14, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13923, pp. 53–68. Springer (2023). https://doi.org/10.1007/978-3-031-39784-4_4
6. Avanzini, M., Dal Lago, U., Ghyselen, A.: Type-based complexity analysis of probabilistic functional programs. In: 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–13 (2019). https://doi.org/10.1109/LICS.2019.8785725
7. Avanzini, M., Dal Lago, U., Yamada, A.: On probabilistic term rewriting. Sci. Comput. Program. **185** (2020)
8. Backhouse, R.C., de Bruin, P.J., Hoogendijk, P.F., Malcolm, G., Voermans, E., van der Woude, J.: Polynomial relators (extended abstract). In: Nivat, M., Rattray, C., Rus, T., Scollo, G. (eds.) Algebraic Methodology and Software Technology (AMAST '91), Proceedings of the Second International Conference on Methodology and Software Technology, Iowa City, USA, 22-25 May 1991. pp. 303–326. Workshops in Computing, Springer (1991)
9. van Bakel, S., Barbanera, F., de'Liguoro, U.: Intersection types for the lambda-mu calculus. Log. Methods Comput. Sci. **14**(1) (2018). https://doi.org/10.23638/LMCS-14(1:2)2018, https://doi.org/10.23638/LMCS-14(1:2)2018
10. Barr, M.: Relational algebras. Lect. Notes Math. **137**, 39–55 (1970)
11. Bird, R.S., de Moor, O.: Algebra of programming. Prentice Hall International series in computer science, Prentice Hall (1997)
12. Bono, V., Dezani-Ciancaglini, M.: A tale of intersection types. In: Hermanns, H., Zhang, L., Kobayashi, N., Miller, D. (eds.) LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020. pp. 7–20. ACM (2020). https://doi.org/10.1145/3373718.3394733, https://doi.org/10.1145/3373718.3394733
13. Breuvart, F., Dal Lago, U.: On intersection types and probabilistic lambda calculi. In: Sabel, D., Thiemann, P. (eds.) Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018. pp. 8:1–8:13. ACM (2018). https://doi.org/10.1145/3236950.3236968, https://doi.org/10.1145/3236950.3236968
14. Carboni, A., Kelly, G.M., Wood, R.J.: A 2-categorical approach to change of base and geometric morphisms i. Cahiers de Topologie et Géométrie Différentielle Catégoriques **32**(1), 47–95 (1991)
15. Clementino, M.M., Hofmann, D., Janelidze, G.: The monads of classical algebra are seldom weakly cartesian. Journal of Homotopy and Related Structures **9**(1), 175–197 (2014)

16. Coppo, M., Dezani-Ciancaglini, M.: A new type assignment for λ-terms. Arch. Math. Log. **19**(1), 139–156 (1978). https://doi.org/10.1007/BF02011875
17. Coppo, M., Dezani-Ciancaglini, M., Honsell, F., Longo, G.: Extended type structures and filter lambda models. In: Lolli, G., Longo, G., Marcja, A. (eds.) Logic Colloquium 82. pp. 241–262. North-Holland, Amsterdam, the Netherlands (1984)
18. Dal Lago, U., Faggian, C., Ronchi Della Rocca, S.: Intersection types and (positive) almost-sure termination. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021). https://doi.org/10.1145/3434313
19. Dal Lago, U., Gavazzo, F., Levy, P.B.: Effectful applicative bisimilarity: Monads, relators, and howe's method. In: Proc. of LICS 2017. pp. 1–12 (2017)
20. Dal Lago, U., Gavazzo, F., Tanaka, R.: Effectful applicative similarity for call-by-name lambda calculi. In: Monica, D.D., Murano, A., Rubin, S., Sauro, L. (eds.) Joint Proceedings of the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic co-located with the 2017 IEEE International Workshop on Measurements and Networking (2017 IEEE M&N), Naples, Italy, September 26-28, 2017. CEUR Workshop Proceedings, vol. 1949, pp. 87–98. CEUR-WS.org (2017), http://ceur-ws.org/Vol-1949/ICTCSpaper06.pdf
21. Dal Lago, U., Gavazzo, F., Tanaka, R.: Effectful applicative similarity for call-by-name lambda calculi. Theor. Comput. Sci. **813**, 234–247 (2020). https://doi.org/10.1016/j.tcs.2019.12.025, https://doi.org/10.1016/j.tcs.2019.12.025
22. Dal Lago, U., Grellois, C.: Probabilistic termination by monadic affine sized typing. ACM Trans. Program. Lang. Syst. **41**(2), 10:1–10:65 (2019). https://doi.org/10.1145/3293605, https://doi.org/10.1145/3293605
23. Davies, R., Pfenning, F.: Intersection types and computational effects. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00). pp. 198–208. ACM (2000). https://doi.org/10.1145/351240.351259
24. de Carvalho, D.: Execution time of λ-terms via denotational semantics and intersection types. Math. Str. in Comput. Sci. **28**(7), 1169–1203 (2018). https://doi.org/10.1017/S0960129516000396
25. de'Liguoro, U., Treglia, R.: The untyped computational λ-calculus and its intersection type discipline. Theor. Comput. Sci. **846**, 141–159 (2020). https://doi.org/10.1016/j.tcs.2020.09.029, https://doi.org/10.1016/j.tcs.2020.09.029
26. de'Liguoro, U., Treglia, R.: Intersection types for a λ-calculus with global store. In: Veltri, N., Benton, N., Ghilezan, S. (eds.) PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021. pp. 5:1–5:11. ACM (2021). https://doi.org/10.1145/3479394.3479400, https://doi.org/10.1145/3479394.3479400
27. de'Liguoro, U., Treglia, R.: From semantics to types: The case of the imperative λ-calculus. vol. 973, p. 114082 (2023). https://doi.org/10.1016/j.tcs.2023.114082, https://doi.org/10.1016/j.tcs.2023.114082
28. Dezani-Ciancaglini, M., Giannini, P., Venneri, B.: Intersection types in java: Back to the future. In: Margaria, T., Graf, S., Larsen, K.G. (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 11200, pp. 68–86. Springer (2018). https://doi.org/10.1007/978-3-030-22348-9_6, https://doi.org/10.1007/978-3-030-22348-9_6
29. Dezani-Ciancaglini, M., Ronchi Della Rocca, S.: Intersection and Reference Types. In: Reflections on Type Theory, Lambda Calculus, and the Mind. pp. 77–86. Radboud University Nijmegen (2007), http://www.di.unito.it/~dezani/papers/dr.pdf

30. Ehrhard, T.: Collapsing non-idempotent intersection types. In: Cégielski, P., Durand, A. (eds.) Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France. LIPIcs, vol. 16, pp. 259–273. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012). https://doi.org/10.4230/LIPIcs.CSL.2012.259, https://doi.org/10.4230/LIPIcs.CSL.2012.259
31. Ehrhard, T., Pagani, M., Tasson, C.: Probabilistic Coherence Spaces are Fully Abstract for Probabilistic PCF. In: Sewell, P. (ed.) The 41th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL14, San Diego, USA. ACM (2014)
32. Faggian, C., Guerrieri, G., de'Liguoro, U., Treglia, R.: On reduction and normalization in the computational core. Mathematical Structures in Computer Science p. 1–48 (2023). https://doi.org/10.1017/S0960129522000433
33. Freeman, T.S., Pfenning, F.: Refinement types for ML. In: Wise, D.S. (ed.) Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991. pp. 268–277. ACM (1991). https://doi.org/10.1145/113445.113468, https://doi.org/10.1145/113445.113468
34. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM **55**(4), 19:1–19:64 (2008). https://doi.org/10.1145/1391289.1391293, https://doi.org/10.1145/1391289.1391293
35. Gautam, N.D.: The validity of equations of complex algebras. Archiv für mathematische Logik und Grundlagenforschung **3**(3), 117–124 (1957)
36. Gavazzo, F., Faggian, C.: A relational theory of monadic rewriting systems, part I. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021. pp. 1–14. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470633, https://doi.org/10.1109/LICS52264.2021.9470633
37. Gavazzo, F., Treglia, R., Vanoni, G.: Monadic intersection types, relationally (extended version) (2024), https://arxiv.org/abs/2401.12744
38. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
39. Hoffman, D., Seal, G.J.: A cottage industry of lax extensions. Categories and General Algebraic Structures with Applications **3**(1), 113–151 (2015)
40. Hofmann, D., Seal, G., Tholen, W.: Monoidal Topology: A Categorical Approach to Order, Metric and Topology. Encyclopedia of Mathematics and its Applications, Cambridge University Press (2014)
41. Hughes, J., Jacobs, B.: Simulations in coalgebra. Theor. Comput. Sci. **327**(1-2), 71–108 (2004)
42. Kawahara, Y.: Notes on the universality of relational functors. Memoirs of the Faculty of Science, Kyushu University. Series A, Mathematics **27**(2), 275–289 (1973)
43. Kelly, G.M.: Basic concepts of enriched category theory. Reprints in Theory and Applications of Categories (10), 1–136 (2005)
44. Kesner, D., Ramos, M., Treglia, R.: A quantitative understanding of exceptions (2023), presented at TLLA 2023
45. Kesner, D., Vial, P.: Non-idempotent types for classical calculi in natural deduction style. Log. Methods Comput. Sci. **16**(1) (2020). https://doi.org/10.23638/LMCS-16(1:3)2020, https://doi.org/10.23638/LMCS-16(1:3)2020
46. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. pp. 416–428. ACM (2009). https://doi.org/10.1145/1480881.1480933

47. Kobayashi, N., Dal Lago, U., Grellois, C.: On the termination problem for probabilistic higher-order recursive programs. Log. Methods Comput. Sci. **16**(4) (2020)
48. Kobayashi, N., Ong, C.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA. pp. 179–188. IEEE Computer Society (2009). https://doi.org/10.1109/LICS.2009.29
49. MacLane, S.: Categories for the Working Mathematician. Springer-Verlag (1971)
50. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89). pp. 14–23. IEEE Computer Society (1989). https://doi.org/10.1109/LICS.1989.39155
51. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
52. Ong, C.L.: On model-checking trees generated by higher-order recursion schemes. In: 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings. pp. 81–90. IEEE Computer Society (2006). https://doi.org/10.1109/LICS.2006.38
53. Pierce, B.C.: Intersection types and bounded polymorphism. Math. Struct. Comput. Sci. **7**(2), 129–193 (1997). https://doi.org/10.1017/S096012959600223X
54. Plotkin, G., Power, J.: Adequacy for algebraic effects. In: Honsell, F., Miculan, M. (eds.) Foundations of Software Science and Computation Structures. pp. 1–24. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
55. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975). https://doi.org/10.1016/0304-3975(75)90017-1
56. Plotkin, G.D., Power, J.: Adequacy for algebraic effects. In: Honsell, F., Miculan, M. (eds.) Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2030, pp. 1–24. Springer (2001). https://doi.org/10.1007/3-540-45315-6_1, https://doi.org/10.1007/3-540-45315-6_1
57. Plotkin, G.D., Power, J.: Notions of computation determine monads. In: FOSSACS 2002. Lecture Notes in Computer Science, vol. 2303, pp. 342–356. Springer (2002). https://doi.org/10.1007/3-540-45931-6_24, https://doi.org/10.1007/3-540-45931-6_24
58. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Appl. Categorical Struct. **11**(1), 69–94 (2003). https://doi.org/10.1023/A:1023064908962
59. Sands, D.: Improvement theory and its applications. In: Gordon, A.D., Pitts, A.M. (eds.) Higher Order Operational Techniques in Semantics, pp. 275–306. Publications of the Newton Institute, Cambridge University Press (1998)
60. Sankappanavar, H.P., Burris, S.: A course in universal algebra. Graduate Texts Math **78** (1981)
61. Schmidt, G.: Relational Mathematics, Encyclopedia of Mathematics and its Applications, vol. 132. Cambridge University Press (2011)
62. Simpson, A., Voorneveld, N.F.W.: Behavioural equivalence via modalities for algebraic effects. ACM Trans. Program. Lang. Syst. **42**(1), 4:1–4:45 (2020). https://doi.org/10.1145/3363518, https://doi.org/10.1145/3363518
63. Stone, M.H.: Postulates for the barycentric calculus. Ann. Mat. Pura Appl. (4) **29**(1), 25–30 (1949). https://doi.org/10.1007/BF02413910

64. Syropoulos, A.: Mathematics of multisets. In: Workshop on Membrane Computing. pp. 347–358. Springer (2000)
65. Tsukada, T., Kobayashi, N.: Complexity of model-checking call-by-value programs. In: Muscholl, A. (ed.) Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8412, pp. 180–194. Springer (2014). https://doi.org/10.1007/978-3-642-54830-7_12
66. Varacca, D.: Probability, nondeterminism and concurrency: two denotational models for probabilistic computation. Ph.D. thesis, Aarhus University (2003)
67. Varacca, D., Winskel, G.: Distributing probability over non-determinism. Math. Struct. Comput. Sci. **16**(1), 87–113 (2006)
68. Vial, P.: Infinitary intersection types as sequences: A new answer to klop's problem. In: LICS. pp. 1–12. IEEE Computer Society (2017)
69. Vial, P.: Every λ-term is meaningful for the infinitary relational model. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 899–908. ACM (2018). https://doi.org/10.1145/3209108.3209133
70. Wadler, P.: Monads for functional programming. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques. Lecture Notes in Computer Science, vol. 925, pp. 24–52. Springer (1995). https://doi.org/10.1007/3-540-59451-5_2
71. Weber, M.: Generic morphisms, parametric representations and weakly cartesian monads. Theory Appl. Categ **13**(14), 191–234 (2004)

Layered Modal Type Theory

Where Meta-programming Meets Intensional Analysis

Jason Z. S. Hu(✉) and Brigitte Pientka

School of Computer Science, McGill University, Montréal, QC, Canada H3A 0E9
`zhong.s.hu@mail.mcgill.ca` `bpientka@cs.mcgill.ca`

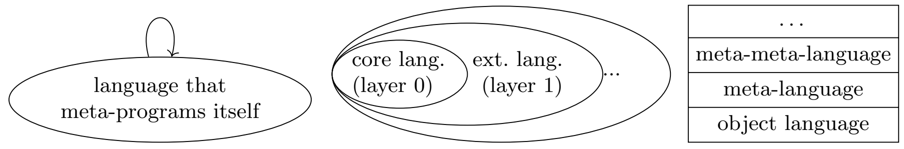
Abstract. We introduce layering to modal type theory to combine type theory with intensional analysis. In particular, we demonstrate this idea by developing a 2-layered modal type theory. At the core of this type theory (layer 0) is a simply typed λ-calculus with no modality. Layer 1 is obtained by extending the core language with one layer of contextual $\square$ types to support pattern matching on potentially open code from layer 0 while retaining normalization. Although both layers fundamentally share the same language and the same typing judgment, we only allow computation at layer 1. As a consequence, layer 0 accurately captures the syntactic representation of code in contrast to the computational behaviors at layer 1. The system is justified by normalization by evaluation (NbE) using a presheaf model. The normalization algorithm extracted from the model is sound and complete and is implemented in Agda.
Layered modal type theory provides a uniform foundation for meta-programming with intensional analysis. We see this work as an important step towards a foundational way to support meta-programming in proof assistants.

Keywords: modal type theory · contextual types · meta-programming · normalization by evaluation· presheaf model

1 Introduction

For the past decades, the problem of combining type theory and meta-programming has been in need for a solution (c.f. [57,15,18,36,47,50,7]). Given the solid and elegant foundations for describing proofs as programs provided by type theories, also supporting meta-programming allows us to think of proof generation as code generation. This opens up the possibility to support proof macros, domain-specific proof generators, proof transformations, and reasoning about meta-programs within the same language.

While support for meta-programming in existing proof assistants is common (e.g. [18,15,61,57]), this is typically achieved via some unverified mechanisms like reflection, requiring significant engineering effort. Moreover, the interplay between these mechanisms and the core type theory is not well-understood, often breaks critical type-theoretic properties like confluence, and lacks theoretical guarantees like normalization. As a consequence, it is often not clear how we



(a) homogeneous style (b) layered style (c) heterogeneous style

Fig. 1: Layered style as a middle ground

can reason about meta-programs themselves. Even guaranteeing that the generated code is well-typed and well-scoped is non-trivial. Hence this leads to a gap between implementations of meta-programming in proof assistants and their theoretical foundations.

Theoretical foundations that combine meta-programming with type-theory typically fall into two categories: the homogeneous style and the heterogeneous style. Homogeneous meta-programming uses a single language capable of meta-programming itself (depicted in Fig. 1a). To provide a logical, type-safe foundation in this style, Davies and Pfenning [17] give a modal λ-calculus with the $\Box$ modality. They use the modal type $\Box T$ to represent the code of type T. Having modal types allows us to differentiate on the type level meta-programs that manipulate code from regular programs in one unified language. Nanevski et al. [39] subsequently extend the modal λ-calculus [17] with contextual types, allowing meta-programming on open code. Nevertheless, the correspondence described by both systems only supports basic primitives like execution and composition of code, but does not suggest a way to support any form of intensional analysis. In fact, supporting intensional analysis in the homogeneous style while retaining properties like confluence and normalization has been fraught with difficulties (c.f. [48]). Most recently, Kavvos [33] notes that we can only soundly extend the modal λ-calculus with intensional analysis for *closed* code if we want to retain confluence. A significant step towards supporting pattern matching on open code in a homogeneous style is taken in Moebius [30]. Moebius is based on System F-style polymorphism. However, its pattern matching does not guarantee coverage. Therefore Moebius does not provide normalization.

In a heterogeneous system, we distinguish between the meta-language and the object language (illustrated by Fig. 1c). Recently, Kovács [36] adapts 2-level type theory (2LTT), originally conceived for homotopy type theory, to dependently typed meta-programming. Here, a dependently typed meta-language sits on top of a less expressive object language. However, this type theory does not support intensional analysis. In contrast, Cocon [47], another 2-level type theory following in the footsteps of previous work [17,39], supports modeling open code and intensional code analysis. Though these heterogeneous systems are modular, this comes at a price: a definition in one level is not directly accessible or reused in the other level. Unlike homogeneous systems, both heterogeneous systems do not support execution of code. Moreover, the separation into two languages leads to two separate investigations of meta-theoretic properties for two languages and ultimately two separate normalization arguments. How to elegantly scale these languages to multiple layers is not obvious, or at least very tedious.

In this paper, we propose a novel *layered* style as a schema to combine meta-programming and type theory (see Fig. 1b) and to combine the advantages of homogeneous and heterogeneous styles. Specifically, our layered modal type theory achieves three features: ① a *run* primitive, which extracts a term of type A given code of type A for all A; ② a *normalizing* type theory; ③ pattern matching on code, which is the most general form of intensional analysis. As a demonstration, we develop a layered modal simple type theory achieving these features. In this type theory, there are a fixed number of layers of languages. The type theory is uniform in the sense that all layers fundamentally *share* a common syntax for their languages and the same typing judgment as in the homogeneous style. Therefore, our layered system has a natural *run* primitive as all homogeneous systems. Furthermore, our layered system follows the matryoshka principle: the language at layer i is *contained* in its meta-language at layer $i+1$. What is added to layer i at layer $i+1$ is the ability to inspect and analyze code from the language at layer i. This matryoshka structure of layers of languages not only ensures uniformity in the syntax and the typing judgment of the type theory, but also provides extra flexibility in distinguishing computational behaviors at different layers. As a principle, we only allow β and η equivalence at the highest layer, so all lower layers are treated as static code which is only identified by its syntax. Layering allows us to encode different computational behaviors at different layers using the same set of equivalence rules. This is crucial to enable sound intensional analysis and establish normalization.

To introduce layering succinctly, we focus on a 2-layered modal simple type theory in this paper. In this 2-layered system, its core language at layer 0 is a simply typed λ-calculus (STLC). At layer 1, STLC is then extended with one layer of meta-programming with the $\square$ modality. The meta-language at layer 1 can only manipulate and analyze code from layer 0, but *not* from its own layer. Following our previous discussion, we only allow computation on layer 1, and terms at layer 0 are treated as pure syntax. This allows us to cleanly define covering pattern matching on code and eventually leads to an elegant normalization proof using a presheaf model.

Summary of Contributions:

1. We develop a 2-layered modal type theory (Sec. 3) which supports running code (feature ①). To prove normalization, we extend the classic presheaf model for STLC [5] to our type theory (Sec. 4). From this presheaf model, we extract its normalization algorithm that is complete and sound.
2. We extend the previous 2-layered modal type theory with pattern matching on code (Sec. 5). We adapt our previous presheaf model to support pattern matching on code and prove that the extracted algorithm is both complete and sound. Thus we achieve features ② and ③.
3. We outline three different dimensions to extend layered modal type theory in Sec. 6. In particular, we discuss extensions to richer systems like System F and Martin-Löf type theory. We also discuss how to extend the expressive power of the computational layer with additional operations, and how to scale our 2-layered system to n layers.

We believe that layering is versatile enough to be adapted to complex systems like System F and Martin-Löf type theory. As such, it provides a systematic way of supporting intensional analysis while retaining normalization. It is a significant step towards closing the gap between implementations that support meta-programming in practice and their theoretical foundations. Interested readers could find more details in our technical report [27] and our Agda code [28].

2 Example Programs in 2-layered Modal Type Theory

In this section, we show how to write and improve the well-known `power` function in layered modal type theory by gradually introducing more features. In general, many common meta-programs including the `power` function use only two layers.

2.1 A Layered Power Function

The `power` function defined by [17, Sec. 3.4] is a classic meta-program and we can define it in our 2-layered type theory with the help of contextual types:

```
power : Nat → □ (x : Nat ⊢ Nat)
power zero     = box (x. 1)
power (succ n) = letbox u ← power n in box (x. u[x/x] * x)
```

In the examples in this section, we use a front-end syntax similar to Haskell and Agda. For clarity, we abbreviate `succ ... (succ zero)` as numbers, e.g. `1` is notation for `succ zero`. The return type of this meta-function is a contextual type `□ (x : Nat ⊢ Nat)`. This type denotes code of type `Nat` with an open variable `x` of type `Nat`. In general, the number of open variables is arbitrary. In the body, we recurse on the input number. If it is `zero`, then the generated code is just `1`. The open variable `x` is not used. In the `succ` case, we first perform the recursive call `power n`. The eliminator `letbox` binds a new *global variable* `u` to an open type `(x : Nat ⊢ Nat)`. We say that `u` has type `Nat` with an open variable `x` of type `Nat`. A global variable is a placeholder for code. It remains visible under a `box` constructor. Regular variables like `n`, on the other hand, cannot directly participate in code construction, so they are hidden inside `box`. When we refer to `u` in `box`, we must instantiate the open variable `x` of `u`. In this case, an identity substitution `[x/x]` suffices. Now `u[x/x]` stands for the `n`'th power of `x` and we obtain our goal by multiplying it with an extra `x`. Our implementation of the `power` function is almost as expected except for the dangling `1`:

```
power 1 = box (x. 1 * x)        power 2 = box (x. (1 * x) * x)
```

We would like to remove the `1`'s because it is the unit element of multiplication. We will make this improvement in the next subsection. Nevertheless, we can already *run* the current code, which is critical for a meta-programming system:

```
letbox u ← power 2 in λ x. u[x/x] : Nat → Nat
```

generates a regular function computing squares. We can also directly run the code with a specific argument:

```
letbox u ← power 2 in u[5/x] = 25
```

would substitute 5 for x and give 25, the square of 5.

2.2 Pattern Matching for Intensional Analysis

An easy way to improve the previous implementation is to *pattern match* on the resulting code and remove all occurrences of 1. However, supporting pattern matching on code in a type-theoretic setting has been notoriously difficult. Previous attempts in the homogeneous style fail to retain the normalization property. To illustrate, consider the intensional isapp function [33,19]. This function simply looks at the structure of a code and returns true if this code is a function application, or false otherwise. Note that isapp's behavior purely depends on the syntactic structure of its argument. In our 2-layered system, this function can be implemented by a pattern matching on code:

```
isapp : □ ( ⊢ Nat) → Bool
isapp x = match x with | ?u ?u' ⇒ true | _ ⇒ false
```

We use pattern matching to inspect the input code x. In our first branch, we return true if x is some function application. Here, ?u and ?u' are both *pattern variables*. We use question marks to distinguish pattern variables and constants, e.g. zero and succ which are the constructors of Nat. This distinction is only necessary in the patterns, and we do not write a question mark when we refer to a pattern variable in the body of the branch. We also omit writing the local context in which the pattern is sensible because it is determined by the type of x. The pattern variables u and u' capture the code of the function and the argument respectively if x is a function application. As they are not used, we could also have written _ _ instead. The other branches are captured by the wildcard and all return false. Let us see how this function behaves:

```
isapp (box ((λ x. x) 10)) = true
isapp (box 10)            = false
```

Kavvos [33] points out that Gabbay and Nanevski's [19] evaluation of isapp is not confluent. It is possible to evaluate the same program in different orders and obtain two different values. For some well-typed code t and s,

```
  letbox u ← box (t s) in isapp (box u)
= isapp (box (t s))                      = true
  letbox u ← box (t s) in isapp (box u)
= letbox u ← box (t s) in false          = false
```

In the second execution, isapp (box u) is evaluated first, and then the overall result is false. In our system, this confluence issue is avoided by preventing the execution of isapp (box u) until it is known what u stands for. This treatment ensures that isapp is stable under substitutions. Hence, the program only evaluates to true. This is a subtle but critical design decision which ultimately enables sound intensional analysis and normalization. We explain more in Sec. 5.2.

With sound pattern matching on code, a simple arithmetic simplifier is implemented to remove the redundant 1's in the previous subsection:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp y = match y with
         | 1 * ?u    ⇒ box (x. u[x/x])
         | ?u * ?u' ⇒ letbox u1 = simp (box (x. u[x/x]))
                        in box (x. u1[x/x] * u'[x/x])
         | _         ⇒ y
```

In the first case, we remove `1` from the multiplication. In the second case, we recursively simplify the first factor. We know this is sufficient because `1` only occurs in the leftmost factor. In the last case, we do not optimize. Since pattern matching is covering, we must either specify all cases or give a wildcard case. At last, we provide a wrapper function `power'`, where we invoke `simp` to simplify the code generated by `power`:

```
power' : Nat → □ (x : Nat ⊢ Nat)
power' n = simp (power n)
```

The `power'` function precisely does what we expect:

```
power' 1 = box (x. x)              power' 2 = box (x. x * x)
```

This example shows that we have full control over code via pattern matching on code, while running `power'` still gives the same behaviors as `power`.

3 A 2-Layered Modal Type Theory

In this section, we introduce a 2-layered modal type theory, which is simple yet powerful enough for many interesting programs like the unoptimized `power` function in the previous example. This system provides a starting point and a basis for a clear understanding of the impact of layering on syntax and semantics. We build a semantic framework for 2-layered modal type theory which is further extended with pattern matching on code in Sec. 5.

2-layered modal type theory is defined as follows:

$$
\begin{array}{llll}
S, T & := & \mathtt{Nat} \mid \Box(\Gamma \vdash T) \mid S \longrightarrow T & \text{(Types, } \mathsf{Typ}\text{)} \\
x, y & & & \text{(Local variables)} \\
u & & & \text{(Global variables)} \\
s, t & := & x \mid u^{\delta} \mid \mathsf{zero} \mid \mathsf{succ}\ t \mid \mathsf{rec}_T\ s\ (x\ y.s')\ t & \text{(Terms, } \mathsf{Exp}\text{)} \\
 & & \mid \mathtt{box}\ t \mid \mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t \mid \lambda x.t \mid s\ t & \\
\delta & := & \cdot \mid \delta, t/x & \text{(Local substitutions)} \\
\Gamma, \Delta & := & \cdot \mid \Gamma, x : T & \text{(Local contexts)} \\
\Phi, \Psi & := & \cdot \mid \Phi, u : (\Gamma \vdash T) & \text{(Global contexts)}
\end{array}
$$

We assume de Bruijn indices as our name representation for convenience but our development generalizes. We use natural numbers `Nat` as a base type. We can construct zero and succ of another `Nat`. $\mathsf{rec}_T\ s\ (x\ y.s')\ t$ is the recursor for `Nat`, where t is the scrutinee, s is the base case and s' is the step case, where x is the predecessor and y is the result from the recursive call. As the recursor for natural numbers is standard, we leave its discussion in the technical report [27].

A function is introduced by λ-abstraction and can be applied to an argument. $\square(\Gamma \vdash T)$ is a contextual type. It stands for code open in context Γ. The `box` constructor introduces terms of type $\square(\Gamma \vdash T)$ and `letbox` is the eliminator for it. We defer our discussion on pattern matching on code to Sec. 5.

For layered systems, we keep track of as many contexts as the layers. These contexts are contained in a fixed-sized *context array* in the judgments. With two layers, a context array only has two contexts $\Phi; \Gamma$. It hence defines a dual-context type theory. Following Pfenning and Davies [42,17], Γ is referred to as a *local context* and its variables are *local variables*, ranged over by x and y. Φ is a *global context* and contains *global variables*, ranged over by u. For a global binding $u : (\Gamma \vdash T)$, we say that u represents code of type T with an open context Γ.

When writing meta-programs, we conceptually distinguish between programs that are dynamic and compute, and code that is static and syntactic. In a homogeneous system, this distinction is captured by types, i.e. program t has type T while code has type $\square(\Gamma \vdash T)$. However, a term t itself does not provide information about whether it is inside of a `box` (hence treated as code), or outside of a `box` (hence a program). For example, only knowing that `succ zero` has type `Nat` does not reveal whether it is a piece of code or a program. The typing judgment for homogeneous systems like $\Psi; \Gamma \vdash t : T$ [42,17] only provides typing information, and does not a priori determine whether t should be considered as code or as a program. Even though one major advantage of a homogeneous system is to use the same language for code and programs, this lack of information is the critical reason for the challenges that we face when combining type theory and intensional analysis.

Layered modal type theory makes the distinction between code and programs explicit. In the typing judgment $\Psi; \Gamma \vdash_i t : T$, we use the subscript $i \in [0, 1]$ to identify the layer at which t is well-typed. This judgment states that the term t has type T *at layer* i. When $i = 0$, t is code and does not compute, and when $i = 1$, t is a program and therefore has rich reduction behaviors. There are three important implications of layering:

1. we can control what types are valid at each layer,
2. we can control what terms are well-typed at each layer, and
3. we can control what terms are equivalent at each layer.

In the first part, we control the validity of types using the validity predicate. In the rules below, we rule out the use of $\square$ at layer 0 and limit layer 1 to at most one layer of $\square$:

$$\frac{}{\texttt{Nat wf}^i} \qquad \frac{S\ \texttt{wf}^i \qquad T\ \texttt{wf}^i}{S \longrightarrow T\ \texttt{wf}^i} \qquad \frac{\Gamma\ \texttt{wf}^0 \qquad T\ \texttt{wf}^0}{\square(\Gamma \vdash T)\ \texttt{wf}^1}$$

This validity predicate only limits the depth of nested $\square$s. Therefore, $(\square(\cdot \vdash \texttt{Nat}) \to \square(\cdot \vdash \texttt{Nat}))\ \texttt{wf}^1$ holds although it has two $\square$s. $\square(\cdot \vdash \square(\cdot \vdash \texttt{Nat}))\ \texttt{wf}^1$ does not hold, because it has two nested layers of $\square$. Moreover, the validity judgment only provides an upper bound, so both $\texttt{Nat wf}^0$ and $\texttt{Nat wf}^1$ hold. This predicate generalizes to $\Psi\ \texttt{wf}^i$ and $\Gamma\ \texttt{wf}^i$ by requiring all types in Ψ and Γ

$\boxed{\Psi; \Gamma \vdash_i t : T}$ and $\boxed{\Psi; \Gamma \vdash_i \delta : \Delta}$ Term t and local substitution δ are well-typed, respectively, in context Ψ and Γ at layer i where $i \in [0, 1]$

$$\frac{\Psi\ \mathtt{wf}^0 \quad \Gamma\ \mathtt{wf}^i}{\Psi; \Gamma \vdash_i \cdot : \cdot} \qquad \frac{\Psi; \Gamma \vdash_i \delta : \Delta \quad \Psi; \Gamma \vdash_i t : T}{\Psi; \Gamma \vdash_i \delta, t/x : \Delta, x : T} \qquad \frac{\Psi; \Gamma \vdash_i \delta : \Delta \quad u : (\Delta \vdash T) \in \Psi}{\Psi; \Gamma \vdash_i u^\delta : T}$$

$$\frac{\Psi\ \mathtt{wf}^0 \quad \Gamma\ \mathtt{wf}^i \quad x : T \in \Gamma}{\Psi; \Gamma \vdash_i x : T} \qquad \frac{\Psi\ \mathtt{wf}^0 \quad \Gamma\ \mathtt{wf}^i}{\Psi; \Gamma \vdash_i \mathsf{zero} : \mathtt{Nat}} \qquad \frac{\Psi; \Gamma \vdash_i t : \mathtt{Nat}}{\Psi; \Gamma \vdash_i \mathsf{succ}\ t : \mathtt{Nat}}$$

$$\frac{\Psi; \Gamma, x : S \vdash_i t : T}{\Psi; \Gamma \vdash_i \lambda x.t : S \longrightarrow T} \qquad \frac{\Psi; \Gamma \vdash_i t : S \longrightarrow T \quad \Psi; \Gamma \vdash_i s : S}{\Psi; \Gamma \vdash_i t\ s : T}$$

$\boxed{\Psi; \Gamma \vdash_i t \approx t' : T}$ Term t and t' are equivalent in contexts Ψ and Γ at layer i

$$\frac{\Psi; \Gamma, x : S \vdash_1 t : T \quad \Psi; \Gamma \vdash_1 s : S}{\Psi; \Gamma \vdash_1 (\lambda x.t)\ s \approx t[s/x] : T} \qquad \frac{\Psi; \cdot \vdash_0 s : T \quad \Psi, u : T; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathtt{letbox}\ u \leftarrow \mathtt{box}\ s\ \mathtt{in}\ t \approx t[s/u] : T'}$$

$$\frac{\Psi; \Gamma \vdash_1 t : S \longrightarrow T}{\Psi; \Gamma \vdash_1 t \approx \lambda x.(t\ x) : S \longrightarrow T}$$

Fig. 2: Typing and equivalence judgments

to comply with the predicate. The validity predicates satisfy the lifting property:

Lemma 1 (Type lifting). *If T wf^0, then T wf^1; if Γ wf^0, then Γ wf^1.*

The lifting property characterizes the matryoshka principle for types and the diagram in Fig. 1b, and says that types and contexts at a lower layer are included at a higher layer.

The fact that the validity predicate only allows $\square$ at layer 1, suggests that its constructor `box` and eliminator `letbox` should also only appear at layer 1, while terms of types `Nat` and functions should appear at both layers. Having a layer in the typing judgment allows us to cleanly restrict valid terms at each layer:

$$\frac{\Gamma\ \mathtt{wf}^1 \quad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t : \square(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma \vdash_1 s : \square(\Delta \vdash T) \quad \Psi, u : (\Delta \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t : T'}$$

`box` t is well-typed at layer 1, only if the code t is well-typed at layer 0. Now a clear line is drawn between code and programs: code lives at layer 0 while programs live at layer 1. The rule for `letbox` is only available at layer 1. The body is type-checked in an extended global context with a new global variable. This global variable is a placeholder for the code computed by s.

The rules are given in Fig. 2. The rules for terms coming from STLC, i.e. zero, succ, λ and function applications, are standard and valid at both layers. Given a term of type `Nat` or a function, we know whether it is code or a program by checking the layer it lives at. Extra validity predicates are added to the premises of the local variable rule and the zero rule to enforce the coherence between terms and types at layer i. Notice that terms from STLC can extend the local context via λ regardless of layers and they can only refer to but not introduce global variables. When referring to a global variable u, a local substitution δ is needed to replace all variables in the local context Δ, as specified by the superscript. The coherence between terms and types requires terms at layer i to have types at the same layer. This criterion is formulated by the following lemma:

Lemma 2 (Syntactic validity). *If $\Psi; \Gamma \vdash_i t : T$, then Ψ wf^0, Γ wf^i and T wf^i for $i \in [0, 1]$.*

Ψ is always valid at layer 0 because it is a context for code from layer 0.

The layer i in the typing judgment $\Psi; \Gamma \vdash_i t : T$ effectively leads to the encapsulation of two languages in the same system. When $i = 0$, only terms in STLC are well-typed, so we work in STLC, and hence Ψ $\texttt{wf}^0$ and Γ $\texttt{wf}^0$ hold. The typing rules ensure that we cannot write any meta-program at this layer and no $\square$ is involved. When $i = 1$, one layer of $\square$ is allowed in addition to STLC. At this layer, we can not only write regular STLC programs, but also write meta-programs that generate STLC programs through $\square$. Thus, we work with a meta-language and an extension of STLC. In this case, Ψ $\texttt{wf}^0$ and Γ $\texttt{wf}^1$ hold. Using layers, we fit both code (layer 0) and programs (layer 1) in a unified set of typing rules and arrive at a middle ground between homogeneous and heterogeneous styles. Code at layer 0 can be lifted to layer 1 and turned into a program. The resulting program is well-typed, due to the following lemma:

Lemma 3 (Term lifting). *If $\Psi; \Gamma \vdash_0 t : T$, then $\Psi; \Gamma \vdash_1 t : T$.*

The lifting property of well-typed terms has two indications. ① A language at layer 0 is *contained* at layer 1. This is the critical intuition of the matryoshka principle and the idea of layering. ② Though a term at layer 0 is code and static, its computational behaviors are recovered by lifting it to layer 1. The second point is what guarantees a universal *run* primitive for all code that is crucial for a meta-programming system and achieves the feature ① in Sec. 1. The term lifting behavior can be trigger by the β rule for $\square$. For some well-typed terms t and s at layer 0 and a local substitution δ that does not refer to u:

$$\texttt{letbox}\ u \leftarrow \texttt{box}\ (\lambda x.t)\ s\ \texttt{in}\ u^{\delta} \approx ((\lambda x.t)\ s)[\delta] \approx t[s/x][\delta]$$

Due to the β rule, u is replaced by $(\lambda x.t)\ s$. The layer-0 term $(\lambda x.t)\ s$ is then lifted to layer 1 on the right hand side and computes. Thus its computational behavior is revived and it is further reduced to $t[s/x]$.

At last, due to layering in the typing rules, the equivalence rules are also layered. There are three groups of equivalence rules: the PER rules which include symmetry and transitivity, congruence rules which are naturally derived from the typing rules, and the computation rules which describe β and η equivalence. The PER and congruence rules apply to all layers, but the computation rules only apply to layer 1. We show all the β and η rules at the bottom of Fig. 2. The PER and congruence rules are standard. $[s/x]$ and $[s/u]$ are local and global substitutions, respectively. They substitute s for x and for u everywhere as expected. The lack of computation at layer 0 ensures that terms at layer 0 are identified *only* by their syntactic structures and indeed behave as code:

Lemma 4 (Static code). *If $\Psi; \Gamma \vdash_0 t \approx s : T$, then $t = s$.*

$$\frac{}{\varepsilon : \cdot \Longrightarrow_g \cdot} \qquad \frac{\gamma : \Psi \Longrightarrow_g \Phi \qquad \Gamma\ \texttt{wf}^0 \qquad T\ \texttt{wf}^0}{q(\gamma) : \Psi, u : (\Gamma \vdash T) \Longrightarrow_g \Phi, u : (\Gamma \vdash T)}$$

$$\frac{\gamma : \Psi \Longrightarrow_g \Phi \qquad \Gamma\ \texttt{wf}^0 \qquad T\ \texttt{wf}^0}{p(\gamma) : \Psi, u : (\Gamma \vdash T) \Longrightarrow_g \Phi} \qquad \frac{}{\varepsilon : \cdot \Longrightarrow_l \cdot} \qquad \frac{\tau : \Gamma \Longrightarrow_l \Delta \qquad T\ \texttt{wf}^1}{q(\tau) : \Gamma, x : T \Longrightarrow_l \Delta, x : T}$$

$$\frac{\tau : \Gamma \Longrightarrow_l \Delta \qquad T\ \texttt{wf}^1}{p(\tau) : \Gamma, x : T \Longrightarrow_l \Delta} \qquad \frac{\gamma : \Psi \Longrightarrow_g \Phi \qquad \tau : \Gamma \Longrightarrow_l \Delta}{\gamma; \tau : \Psi; \Gamma \Longrightarrow \Phi; \Delta}$$

Fig. 3: Global and local weakenings

This lemma justifies our treatment of terms at layer 0 as code and prepares for the addition of pattern matching on code in Sec. 5.

Finally, we specify global substitutions between global contexts. Global substitutions are defined in the usual way as lists of terms:

$$\sigma := \cdot \mid \sigma, t/u \qquad \text{(Global substitutions)}$$

Due to layering, all terms in a global substitution must live at layer 0:

$$\frac{\Psi\ \texttt{wf}^0}{\Psi \vdash \cdot : \cdot} \qquad \frac{\Psi \vdash \sigma : \Phi \qquad \Psi; \Gamma \vdash_0 t : T}{\Psi \vdash \sigma, t/u : \Phi, u : (\Gamma \vdash T)}$$

Given a global substitution σ, we can apply it to a term:

$$\begin{array}{rcll}
x[\sigma] & := & x & \\
u^\delta[\sigma] & := & \sigma(u)[\delta[\sigma]] & \text{(lookup } u \text{ in } \sigma) \\
\mathsf{zero}[\sigma] & := & \mathsf{zero} & \\
\mathsf{succ}\ t[\sigma] & := & \mathsf{succ}\ (t[\sigma]) & \\
\lambda x.t[\sigma] & := & \lambda x.(t[\sigma]) & \\
s\ t[\sigma] & := & (s[\sigma])\ (t[\sigma]) & \\
\texttt{box}\ t[\sigma] & := & \texttt{box}\ (t[\sigma]) & \\
\texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t[\sigma] & := & \texttt{letbox}\ u \leftarrow s[\sigma]\ \texttt{in}\ (t[\sigma, u/u]) &
\end{array}$$

where $\delta[\sigma]$ applies σ to all terms in δ. Global substitutions do not handle local variables, so in the case of local variables we just return x, while in the case of global variables we look up σ and apply the globally substituted local substitution $\delta[\sigma]$ to the result of the lookup. σ propagate in most cases recursively. In the case of `letbox`, we extend the substitution and apply $\sigma, u/u$ to the body t. Global substitutions compose and have identity. We write $\Psi \vdash \mathsf{id}_\Psi : \Psi$, and often omit the subscript whenever it can be inferred.

4 Presheaf Model and Normalization by Evaluation

We now establish normalization by evaluation (NbE) [37,11,1] of the 2-layered modal type theory. NbE is a technique to establish the normalization property. An NbE proof usually proceeds in two steps: first, we evaluate terms of a type

$$\begin{aligned} [\![\_]\!] &: \mathsf{Typ} \to \mathcal{W}^{op} \Longrightarrow \mathcal{S}et \\ [\![\mathtt{Nat}]\!] &:= \mathsf{Nf}^{\mathtt{Nat}} \\ [\![\square(\Gamma \vdash T)]\!] &:= \mathsf{Nf}^{\square(\Gamma \vdash T)} \\ [\![S \longrightarrow T]\!] &:= [\![S]\!] \hat{\longrightarrow} [\![T]\!] \end{aligned} \qquad \begin{aligned} [\![\Phi]\!]^0_{\Psi} &:= \{\gamma \mid \gamma : \Psi \Longrightarrow_g \Phi\} \\ [\![\Phi]\!]^1_{\Psi} &:= \{\sigma \mid \Psi \vdash \sigma : \Phi\} \\ [\![\cdot]\!]_{\Psi;\Gamma} &:= \{*\} \\ [\![\Delta, x : T]\!]_{\Psi;\Gamma} &:= [\![\Delta]\!]_{\Psi;\Gamma} \times [\![T]\!]_{\Psi;\Gamma} \end{aligned}$$

Fig. 4: Interpretations of types, and global and local contexts

theory into some chosen domain; second, normal forms are extracted from values in this domain. Our chosen domain is a presheaf category. A presheaf category is a functor category from some base category to the category of sets. A carefully chosen base category leads to an intuitive normalization proof. In this section, we use the category of weakenings as the base category. The presheaf model shown here is a moderate extension of the classic presheaf model of STLC [5].

4.1 Category of Weakenings

In the category of weakenings, the objects are the dual contexts and morphisms are weakenings between dual contexts. Weakenings between dual contexts are just tuples of global and local weakenings. They individually are defined in the same way as weakenings in STLC as below:

$$\gamma := \varepsilon \mid q(\gamma) \mid p(\gamma) \quad \text{(Global weakenings)} \quad \tau := \varepsilon \mid q(\tau) \mid p(\tau) \quad \text{(Local weakenings)}$$

Their typing rules are virtually identical with the only difference in the validity predicates (Fig. 3). The q constructor extends a weakening with the same type, while p actually weakens the context. $\Psi \Longrightarrow_g \Phi$ denotes global weakenings and $\Gamma \Longrightarrow_l \Delta$ denotes local weakenings. Then weakenings of dual contexts $\gamma; \tau : \Psi; \Gamma \Longrightarrow \Phi; \Delta$ are tuples of global and local weakenings. Both global and local weakenings have composition and identity. We write id_Ψ and id_Γ for the identity global and local weakenings, respectively. We often omit the subscript when it can be inferred from the context. Identity and composition of weakenings $\Psi; \Gamma \Longrightarrow \Phi; \Delta$ are defined pairwise. We verify that dual contexts and weakenings form a category, which is referred to as $\mathcal{W}$. This is the base category that we will be working with. We sometimes also need to work with $\mathcal{GW}$, the category of global contexts and global weakenings.

4.2 Presheaf Model and Interpretations

In this section, we define the normalization algorithm with $\mathcal{W}$ as our base category. The algorithm normalizes terms to their $\beta\eta$-normal forms, which are defined as follows:

$$\begin{aligned} w &:= v \mid \mathsf{zero} \mid \mathsf{succ}\ w \mid \mathsf{box}\ t \mid \lambda x.w && \text{(Normal form } (\mathsf{Nf})) \\ v &:= x \mid u^{\theta} \mid v\ w \mid \mathsf{rec}_T\ w\ (x\ y.w')\ v \mid \mathtt{letbox}\ u \leftarrow v\ \mathtt{in}\ w && \text{(Neutral form } (\mathsf{Ne})) \\ \theta &:= \cdot \mid \theta, w/x && \text{(Normal local substitutions)} \end{aligned}$$

Notice that `box` t is already normal for any t. This is expected because `box` t regards t as static code so t cannot be reduced. These definitions induce the sets of well-typed normal and neutral forms:

$$\mathsf{Nf}^{T}_{\Psi;\Gamma} := \{w \mid \Psi;\Gamma \vdash_1 w : T\} \qquad \mathsf{Ne}^{T}_{\Psi;\Gamma} := \{v \mid \Psi;\Gamma \vdash_1 v : T\}$$

The sets only capture terms at layer 1 due to the lack of reductions at layer 0. Nf^T and Ne^T then are induced presheaves mapping dual contexts to the sets of normal and neutral forms, respectively.

Next we give the interpretation of types. The interpretation of function types is presheaf exponentials derived from the Yoneda lemma with naturality:

$$\begin{gathered} F \hat{\longrightarrow} G : \mathcal{W}^{op} \Longrightarrow \mathcal{S}et \\ (F \hat{\longrightarrow} G)_{\ \Psi;\Gamma} := \forall\ \gamma;\tau : \Phi;\Delta \Longrightarrow \Psi;\Gamma\ .\ F_{\ \Phi;\Delta} \to G_{\ \Phi;\Delta} \end{gathered}$$

We define the interpretations of types, and global and local contexts in Fig. 4. Both `Nat` and $\square(\Gamma \vdash T)$ are interpreted as their presheaves of normal forms. In particular, $[\![\square(\Gamma \vdash T)]\!]$ is not even recursive. This case effectively interprets $\square(\Gamma \vdash T)$ as the code of T open in Γ. Based on the definition, two possible kinds of terms in $\mathsf{Nf}^{\square(\Gamma\vdash T)}$ are either neutral or of the form `box` t. In the latter case, we have gained access to the syntax of t, permitting more complex operations like pattern matching on code.

The interpretation of global contexts is layered. At layer 1, it is the presheaf of global substitutions, containing code at layer 0 awaiting to be evaluated. At layer 0, it is the Hom set of $\mathcal{GW}$, i.e. the presheaf of global weakenings. This definition is motivated technically to ensure the naturality of evaluation of terms to be defined shortly. We let σ to range over $[\![\Phi]\!]^i_\Psi$ when i is unknown. Local contexts are interpreted as iterated products of values as usual, where $*$ is the unique element of a chosen singleton set. A dual context is interpreted pairwise:

$$[\![\Phi;\Delta]\!]^i{}_{\Psi;\Gamma} := [\![\Phi]\!]^i_\Psi \times [\![\Delta]\!]_{\ \Psi;\Gamma}$$

All interpretations above are functors:

Lemma 5. *$[\![T]\!]$, $[\![\Phi]\!]^i$, $[\![\Delta]\!]$ and $[\![\Phi;\Delta]\!]^i$ are presheaves. $[\![\Phi]\!]^i$ is from $\mathcal{GW}$.*

If $a \in [\![T]\!]_{\ \Phi;\Delta}$ and $\gamma;\tau : \Psi;\Gamma \Longrightarrow \Phi;\Delta$, we write $a[\gamma;\tau]$ for the functorial action of $\gamma;\tau$ on a. We generalize this notation to other functors.

Finally, we define the evaluation functions, interpreting terms as natural transformations between presheaves. This interpretation relies on two other natural transformations, reification and reflection, which map Ne^T to $[\![T]\!]$ and $[\![T]\!]$ to Nf^T, respectively. All four natural transformations are defined in Fig. 5. Since `Nat` and $\square(\Gamma \vdash T)$ are interpreted as presheaves of normal forms, their cases in reification and reflection are just identities. The case for functions is defined in the same way as in STLC.

Our evaluation is a moderate extension of the evaluation of STLC [5]. The evaluation function is layered because the type theory itself is layered. The cases overlapping with STLC are identical, so we only discuss the modal cases. The

$$
\begin{aligned}
\downarrow^{T}_{\Psi;\Gamma} &: [\![T]\!]_{\Psi;\Gamma} \to \mathsf{Nf}^{T}_{\Psi;\Gamma} && \textbf{(Reification)}\\
\downarrow^{\mathtt{Nat}}_{\Psi;\Gamma}(a) &:= a\\
\downarrow^{\Box T}_{\Psi;\Gamma}(a) &:= a\\
\downarrow^{S\longrightarrow T}_{\Psi;\Gamma}(a) &:= \lambda x.\ \downarrow^{T}_{\Psi;\Gamma,x:S}(a\ (\mathsf{id};p(\mathsf{id})\ ,\ \uparrow^{S}_{\Psi;\Gamma,x:S}(x)))\\
&\qquad \text{(where } \mathsf{id};p(\mathsf{id}) : \Psi;\Gamma,x{:}S \Longrightarrow \Psi;\Gamma)\\
\uparrow^{T}_{\Psi;\Gamma} &: \mathsf{Ne}^{T}_{\Psi;\Gamma} \Longrightarrow [\![T]\!]_{\Psi;\Gamma} && \textbf{(Reflection)}\\
\uparrow^{B}_{\Psi;\Gamma}(v) &:= v\\
\uparrow^{\Box T}_{\Psi;\Gamma}(v) &:= v\\
\uparrow^{S\longrightarrow T}_{\Psi;\Gamma}(v) &:=\\
&(\gamma;\tau : \Phi;\Delta \Longrightarrow \Psi;\Gamma)(a \in [\![S]\!]_{\Phi;\Delta}) \mapsto \uparrow^{T}_{\Phi;\Delta}(v[\gamma;\tau]\ \downarrow^{S}_{\Phi;\Delta}(a))\\
[\![\_]\!]^{i}_{\Psi;\Gamma} &: \Phi;\Delta \vdash_i t : T \to [\![\Phi;\Delta]\!]^{i}_{\Psi;\Gamma} \to [\![T]\!]_{\Psi;\Gamma} && \textbf{(Evaluation)}\\
[\![\mathsf{zero}]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) &:= \mathsf{zero}\\
[\![\mathsf{succ}\ t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) &:= \mathsf{succ}\ ([\![t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho))\\
[\![u^{\delta}]\!]^{0}_{\Psi;\Gamma}(\gamma;\rho) &:= \uparrow^{T}_{\Psi;\Gamma}(u[\gamma]^{\theta})\\
&\text{(where } u : (\Delta' \vdash T) \in \Phi,\ \Phi;\Delta \vdash_0 \delta : \Delta', \text{ and } \theta := \downarrow^{\Delta'}_{\Psi;\Gamma}([\![\delta]\!]^{0}_{\Psi;\Gamma}(\gamma;\rho)))\\
[\![u^{\delta}]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) &:= [\![\sigma(u)]\!]^{0}_{\Psi;\Gamma}(\mathsf{id};[\![\delta]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho))\\
[\![x]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) &:= \rho(x) && \text{(lookup } x \text{ in } \rho)\\
[\![\lambda x : S.t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) &:=\\
&(\gamma;\tau : \Phi';\Delta' \Longrightarrow \Psi;\Gamma)(a \in [\![S]\!]_{\Phi';\Delta'}) \mapsto [\![t]\!]^{i}_{\Phi';\Delta'}(\sigma';(\rho',a))\\
&\qquad \text{(where } (\sigma';\rho') := \sigma;\rho[\gamma;\tau] \in [\![\Phi;\Delta]\!]^{i}_{\Phi';\Delta'})\\
[\![t\ s]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) &:= [\![t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho)(\mathsf{id}_{\Psi;\Gamma}, [\![s]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho))\\
[\![\mathtt{box}\ t]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) &:= \mathtt{box}\ (t[\sigma])\\
[\![\mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) &:= [\![t]\!]^{1}_{\Psi;\Gamma}(\sigma, s'/u;\rho) && \text{(if } [\![s]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) = \mathtt{box}\ s')\\
[\![\mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) &:=\\
&\uparrow^{T}_{\Psi;\Gamma}(\mathtt{letbox}\ u \leftarrow v\ \mathtt{in}\ \downarrow^{T}_{\Psi,u:S;\Gamma}([\![t]\!]^{1}_{\Psi,u:S;\Gamma}(\sigma', u^{\mathsf{id}}/u;\rho')))\\
&\text{(if } [\![s]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) = v, \text{ also } (\sigma';\rho') := (\sigma;\rho)[p(\mathsf{id});\mathsf{id}] \in [\![\Phi;\Delta]\!]^{1}_{\Psi,u:S;\Gamma})\\
[\![\_]\!]^{i}_{\Psi;\Gamma} &: \Phi;\Delta \vdash_i \delta : \Delta' \to [\![\Phi;\Delta]\!]^{i}_{\Psi;\Gamma} \to [\![\Delta']\!]_{\Psi;\Gamma}\\
& && \textbf{(Substitution Evaluation)}\\
[\![\cdot]\!]^{i}_{\Psi;\Gamma}(\_) &:= *\\
[\![\delta, t/x]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) &:= ([\![\delta]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho), [\![t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho))
\end{aligned}
$$

Fig. 5: Definitions of reification, reflection and evaluation

$\mathtt{box}\ t$ case is only available at layer 1. In this case, we directly propagate σ under $\mathtt{box}$. In the $\mathtt{letbox}$ case, we first evaluate s. Given $[\![s]\!]^{1}_{\Psi;\Gamma} \in [\![\Box(\Gamma \vdash S)]\!]^{1}_{\Psi;\Gamma} = \mathsf{Nf}^{\Box(\Gamma\vdash S)}_{\Psi;\Gamma}$, this evaluation has two possible results: it returns either a $\mathtt{box}\ s'$, or a neutral v. In the first case, we just recurse with σ extended with s' for u. In the second case of $\mathtt{letbox}$, some neutral v blocks the evaluation, so we can only recurse on the body t with u as is and with σ and ρ properly weakened. To obtain a $[\![T]\!]_{\Psi;\Gamma}$, we reify the evaluation of t and obtain a normal form, using which we obtain a neutral of $\mathtt{letbox}$. A reflection of this neutral gives us a $[\![T]\!]_{\Psi;\Gamma}$.

The interpretation of global variables is the most interesting. When u^{δ} is referred to at layer 1, we are evaluating some code and turning it into a program, i.e. *running* it. We retrieve the code by looking up u in σ, and continue the evaluation at layer 0 with an environment obtained by evaluating δ. Notice that the layer decreases so the interpretation is well-founded regardless of the size of $\sigma(u)$. The evaluation of a local substitution recursively evaluates all terms in the local substitution. If we refer to u^δ at layer 0, then u should stay neutral. Moreover, the evaluation function is required to return a natural transformation. Both requirements lead to the interpretation of global contexts as Hom set of $\mathcal{GW}$ at layer 0, since a weakened global variable is still a global variable and neutral. We first normalize δ by evaluating and then reifying it ($\downarrow^{\Delta'}_{\Psi;\Gamma}$) and obtain a $[\![T]\!]_{\Psi;\Gamma}$ by reflection. Last, reification, reflection and evaluation are all natural transformations:

Lemma 6 (Naturality). *If $\gamma;\tau : \Psi';\Gamma' \Longrightarrow \Psi;\Gamma$,*

- *if $a \in [\![T]\!]_{\Psi;\Gamma}$, then $\downarrow^T_{\Psi;\Gamma}(a)[\gamma;\tau] = \downarrow^T_{\Psi';\Gamma'}(a[\gamma;\tau])$;*
- *if $v \in \mathsf{Ne}^T_{\Psi;\Gamma}$, then $\uparrow^T_{\Psi;\Gamma}(v)[\gamma;\tau] = \uparrow^T_{\Psi';\Gamma'}(v[\gamma;\tau])$;*
- *if $\Phi;\Delta \vdash_i t : T$ and $\sigma;\rho \in [\![\Phi;\Delta]\!]^i_{\Psi;\Gamma}$, then $[\![t]\!]^i_{\Psi;\Gamma}(\sigma;\rho)[\gamma;\tau] = [\![t]\!]^i_{\Psi';\Gamma'}((\sigma;\rho)[\gamma;\tau])$.*

The NbE algorithm is given by composing the interpretations:

Definition 1. *A normalization by evaluation algorithm given $\Psi;\Gamma \vdash_1 t : T$ is*

$$\begin{aligned} \mathsf{nbe}^T_{\Psi;\Gamma}(t) &: \mathsf{Nf}^T_{\Psi;\Gamma} \\ \mathsf{nbe}^T_{\Psi;\Gamma}(t) &:= \downarrow^T_{\Psi;\Gamma}([\![t]\!]^1_{\Psi;\Gamma}(\uparrow^{\Psi;\Gamma})) \end{aligned}$$

where $\uparrow^{\Psi;\Gamma} \in [\![\Psi;\Gamma]\!]^1_{\Psi;\Gamma}$ is a tuple of the identity global substitution and the identity environment.

This algorithm is correct due to the following two theorems:

Theorem 1 (Completeness). *If $\Psi;\Gamma \vdash_1 t \approx t' : T$, then $\mathsf{nbe}^T_{\Psi;\Gamma}(t) = \mathsf{nbe}^T_{\Psi;\Gamma}(t')$.*

Theorem 2 (Soundness). *If $\Psi;\Gamma \vdash_1 t : T$, then $\Psi;\Gamma \vdash_1 t \approx \mathsf{nbe}^T_{\Psi;\Gamma}(t) : T$.*

The completeness theorem states that equivalent terms have equal normal forms, so we can compare the syntactic equality between normal forms to decide whether two terms are equivalent. The soundness theorem states that a well-typed term has and is equivalent to its normal form. Notice that the theorems are about layer 1 because only terms at this layer compute. In the remainder of this section, we outline only the soundness proof. For complete details, please refer to our technical report [27].

4.3 Soundness

The soundness theorem is established via gluing models, which relate syntactic terms with semantic values. In our 2-layered system, we need two layers of gluing models, which reflect the fact that we are actually operating in two languages. For a gluing relation R, we write $a \sim b \in R$ to denote $(a, b) \in R$.

Layer-0 Gluing Model We begin with the gluing model for natural numbers. It recursively relates a term t and a normal form of type $\texttt{Nat}$. This gluing relation applies for both layers:

$$\frac{\Psi; \Gamma \vdash_1 t \approx \mathsf{zero} : \texttt{Nat}}{t \sim \mathsf{zero} \in \texttt{Nat}_{\Psi;\Gamma}} \qquad \frac{\Psi; \Gamma \vdash_1 t \approx \mathsf{succ}\ t' : \texttt{Nat} \quad t' \sim w \in \texttt{Nat}_{\Psi;\Gamma}}{t \sim \mathsf{succ}\ w \in \texttt{Nat}_{\Psi;\Gamma}} \qquad \frac{\Psi; \Gamma \vdash_1 t \approx v : \texttt{Nat}}{t \sim v \in \texttt{Nat}_{\Psi;\Gamma}}$$

At layer 0, for all $T\ \texttt{wf}^0$, its gluing model is:

$$\begin{aligned}
(\!| T |\!)^0_{\Psi;\Gamma} &\subseteq \mathsf{Exp} \times [\![T]\!]_{\Psi;\Gamma} \\
(\!| \texttt{Nat} |\!)^0_{\Psi;\Gamma} &:= \texttt{Nat}_{\Psi;\Gamma} \\
(\!| S \longrightarrow T |\!)^0_{\Psi;\Gamma} &:= \{(t, a) \mid \forall\ \gamma; \tau : \Phi; \Delta \Longrightarrow \Psi; \Gamma, s \sim b \in (\!| S |\!)^0_{\Phi;\Delta} \,. \\
&\qquad\qquad t[\gamma;\tau]\ s \sim a(\gamma;\tau, b) \in (\!| T |\!)^0_{\Phi;\Delta}\}
\end{aligned}$$

$(\!| T |\!)^0$ does not have a case for $\square$ due to $T\ \texttt{wf}^0$. The function case requires that the results of function applications remain related for all weakenings and all related arguments. The gluing between local substitutions and evaluation environments $\delta \sim \rho \in (\!| \Delta |\!)^0_{\Psi;\Gamma}$ is defined by using $(\!| T |\!)^0$ to relate terms and values pairwise.

Definition 2. *We define the semantic judgment at layer* 0*:*

$$\Psi; \Gamma \Vdash_0 t : T := \forall\ \gamma : \Phi \Longrightarrow_g \Psi \textit{ and } \delta \sim \rho \in (\!| \Gamma |\!)^0_{\Phi;\Delta} \,.\, t[\gamma][\delta] \sim [\![t]\!]^0_{\Phi;\Delta}(\gamma; \rho) \in (\!| T |\!)^0_{\Phi;\Delta}$$

The semantic judgment at layer 0 only universally quantifies over global weakenings.

Layer-1 Gluing Model The reason why we must define the layer-0 gluing model first is that we refer to $\Psi; \Gamma \Vdash_0 t : T$ in our layer-1 model. The semantics of $\square(\Delta \vdash T)$ is given in terms of the semantic judgment at layer 0:

$$\frac{\Psi; \Gamma \vdash_1 t \approx \texttt{box}\ s : \square(\Delta \vdash T) \qquad \Psi; \Delta \Vdash_0 s : T}{t \sim \texttt{box}\ s \in \square(\Delta \vdash T)_{\Psi;\Gamma}} \qquad \frac{\Psi; \Gamma \vdash_1 t \approx v : \square(\Delta \vdash T)}{t \sim v \in \square(\Delta \vdash T)_{\Psi;\Gamma}}$$

In the first rule, t is related to $\texttt{box}\ s$ and s is a semantically well-typed term at layer 0. The premise $\Psi; \Delta \Vdash_0 s : T$ is necessary when we prove the semantic typing rule for $\texttt{letbox}$. Without it, we will not able to maintain the semantic well-formedness of global substitutions during evaluation and in the semantic judgment at layer 1. The details are in our technical report. The gluing model at layer 1 for $T\ \texttt{wf}^1$ is now defined as:

$$\begin{aligned}
(\!| T |\!)^1_{\Psi;\Gamma} &\subseteq \mathsf{Exp} \times [\![T]\!]_{\Psi;\Gamma} \\
(\!| \texttt{Nat} |\!)^1_{\Psi;\Gamma} &:= \texttt{Nat}_{\Psi;\Gamma} \\
(\!| \square(\Delta \vdash T) |\!)^1_{\Psi;\Gamma} &:= \square(\Delta \vdash T)_{\Psi;\Gamma} \\
(\!| S \longrightarrow T |\!)^1_{\Psi;\Gamma} &:= \{(t, a) \mid \forall\ \gamma; \tau : \Phi; \Delta \Longrightarrow \Psi; \Gamma, s \sim b \in (\!| S |\!)^1_{\Phi;\Delta} \,. \\
&\qquad\qquad t[\gamma;\tau]\ s \sim a(\gamma;\tau, b) \in (\!| T |\!)^1_{\Phi;\Delta}\}
\end{aligned}$$

Compared to the layer-0 model, the layer-1 model only has an extra case $(\!| \square(\Delta \vdash T) |\!)^1$. The other two cases are just the same:

Lemma 7. *If T wf^0, then $(\!|T|\!)^0_{\Psi;\Gamma} = (\!|T|\!)^1_{\Psi;\Gamma}$.*

This lemma semantically describes the matryoshka principle, witnessing the subsumption of layer 0 into layer 1. The semantic judgment at layer 1 is universally quantified over a semantic global substitution defined below:

$$\frac{\Psi\ \texttt{wf}^0}{\Psi \Vdash \cdot : \cdot} \qquad\qquad \frac{\Psi \Vdash \sigma : \Phi \qquad \Psi; \Gamma \Vdash_0 t : T}{\Psi \Vdash \sigma, t/u : \Phi, u : (\Gamma \vdash T)}$$

We define the semantic judgment at layer 1:

$$\Psi; \Gamma \Vdash_1 t : T := \forall\ \Phi \Vdash \sigma : \Psi \text{ and } \delta \sim \rho \in (\!|\Gamma|\!)^1_{\Phi;\Delta}\ .\ t[\sigma][\delta] \sim [\![t]\!]^1_{\Phi;\Delta}(\sigma;\rho) \in (\!|T|\!)^1_{\Phi;\Delta}$$

The fundamental theorem is established by proving all semantic typing rules:

Theorem 3 (Fundamental). *If $\Psi; \Gamma \vdash_i t : T$, then $\Psi; \Gamma \Vdash_i t : T$.*
If $\Psi; \Gamma \vdash_i \delta : \Delta$, then $\Psi; \Gamma \Vdash_i \delta : \Delta$.

5 Supporting Pattern Matching on Code

In the previous section, we have achieved feature ① and partly feature ②. In this section, we extend the previous system with pattern matching on code, so all features are concluded. We adapt the presheaf model and show that the normalization algorithm remains complete and sound. We introduce a creative semantics in the soundness proof in order to justify pattern matching on code.

5.1 Extension of Pattern Matching

In this section, we extend our previous 2-layered modal type theory with pattern matching on code as follows.

$$\begin{array}{lll}
s, t := & \cdots \mid \mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b} & \text{(Terms, }\mathsf{Exp}\text{)} \\
b \quad := & \mathsf{var}_x \Rightarrow t \mid \mathsf{zero} \Rightarrow t \mid \mathsf{succ}\ ?u \Rightarrow t \mid \lambda x.?u \Rightarrow t \mid ?u\ ?u' \Rightarrow t & \\
& \mid \mathsf{rec}_T\ ?u\ (x\ y.?u')\ ?u'' \Rightarrow t & \text{(Branches)}
\end{array}$$

We extend the system with another elimination form of $\square(\Gamma \vdash T)$, pattern matching ($\mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b}$), where $\overrightarrow{b}$ is a list of *all possible branches* of t. The branches only need to match terms in STLC because pattern matching is only available at layer 1 and the scrutinee is code from layer 0. We do not directly support nested patterns like $(\lambda y.?u)\ ?u'$ to keep the system simple, but they can be encoded as nested pattern matchings. Supporting any useful general recursor (e.g. [47,29]) would require context variables, which abstract over local contexts, and type polymorphism. We see these extensions orthogonal to layering and leave it to future work.

Further modifications to the typing and equivalence rules are in Fig. 6. We omit the case for rec for conciseness. The typing rule for match uses the judgment $\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'$. This judgment enumerates all possible branches based

$\boxed{\Psi; \Gamma \vdash_i t : T}$ Term t has type T in contexts Ψ and Γ at layer i where $i \in [0, 1]$

$$\frac{\Psi; \Gamma \vdash_1 s : \Box(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \mathsf{match}\ s\ \mathsf{with}\ \overrightarrow{b} : T'}$$

$\boxed{\Psi; \Gamma \vdash_1 b : \Delta \vdash T \Rightarrow T'}$ b is a branch of type T' w.r.t. a code of type T open in Δ.

$$\frac{\Delta\ \mathtt{wf}^0 \qquad x : T \in \Delta \qquad \Psi; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{var}_x \Rightarrow t : \Delta \vdash T \Rightarrow T'} \qquad \frac{\Delta\ \mathtt{wf}^0 \qquad \Psi; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{zero} \Rightarrow t : \Delta \vdash \mathtt{Nat} \Rightarrow T'}$$

$$\frac{\Psi, u : (\Delta \vdash \mathtt{Nat}); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{succ}\ ?u \Rightarrow t : \Delta \vdash \mathtt{Nat} \Rightarrow T'} \qquad \frac{\Psi, u : (\Delta, x : S \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \lambda x.?u \Rightarrow t : \Delta \vdash S \longrightarrow T \Rightarrow T'}$$

$$\frac{\forall\ S\ \mathtt{wf}^0 . \Psi, u : (\Delta \vdash S \longrightarrow T), u' : (\Delta \vdash S); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 ?u\ ?u' \Rightarrow t : \Delta \vdash T \Rightarrow T'}$$

$\boxed{\Psi; \Gamma \vdash_i t \approx t' : T}$ Terms t and t' are equivalent (β rules for match)

$$\frac{x : T \in \Delta \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T' \qquad \overrightarrow{b}(x) = \mathsf{var}_x \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ x\ \mathsf{with}\ \overrightarrow{b} \approx t : T'}$$

$$\frac{\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash \mathtt{Nat} \Rightarrow T' \qquad \overrightarrow{b}(\mathsf{zero}) = \mathsf{zero} \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ \mathsf{zero}\ \mathsf{with}\ \overrightarrow{b} \approx t : T'}$$

$$\frac{\Psi; \Delta \vdash_0 s : \mathtt{Nat} \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash \mathtt{Nat} \Rightarrow T' \qquad \overrightarrow{b}(\mathsf{succ}\ s) = \mathsf{succ}\ ?u \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ (\mathsf{succ}\ s)\ \mathsf{with}\ \overrightarrow{b} \approx t[s/u] : T'}$$

$$\frac{\Psi; \Delta, x : S \vdash_0 s : T \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash S \longrightarrow T \Rightarrow T' \qquad \overrightarrow{b}(\lambda x.s) = \lambda x.?u \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ (\lambda x.s)\ \mathsf{with}\ \overrightarrow{b} \approx t[s/u] : T'}$$

$$\frac{\Psi; \Delta \vdash_0 t : S \longrightarrow T \quad \Psi; \Delta \vdash_0 s : S \quad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T' \quad \overrightarrow{b}(t\ s) = ?u\ ?u' \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ (t\ s)\ \mathsf{with}\ \overrightarrow{b} \approx t[t/u, s/u'] : T'}$$

Fig. 6: Adjusted rules with contextual types

on the type of the scrutinee. This guarantees coverage of pattern matching, i.e. that $\overrightarrow{b}$ is indeed a list of *all possible branches* for a given scrutinee of type $\Box(\Delta \vdash T)$.

All typing rules for individual branches are similar. For example, if the pattern is $\lambda x.?u$, then u captures the body of some λ. The branch body t is checked with u bound to $(\Delta, x : S \vdash T)$, which has a larger local context than Δ which we begin with. If the branch matches a function application, our premise requires t is well-typed for all $S\ \mathtt{wf}^0$. This universal quantification should be read as a higher-order derivation that applies for all $S\ \mathtt{wf}^0$ (see also [60]) and where we keep S abstract as a parameter.

The bottom of Fig. 6 are the β rules for pattern matching. Based on the structure of the scrutinee, we dispatch to the right branch and propagate instantiations for pattern variables via global substitutions to the bodies. Notations like $\overrightarrow{b}(\mathsf{succ}\ s)$ denote the lookup of $\overrightarrow{b}$ based on a given shape. For example, $\overrightarrow{b}(\mathsf{succ}\ s) = \mathsf{succ}\ ?u \Rightarrow t$ means that we look up $\mathsf{succ}\ s$ in $\overrightarrow{b}$, and find the branch $\mathsf{succ}\ ?u \Rightarrow t$. Then s is meant to substitute u in t. This lookup is guaranteed to succeed because $\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'$ is covering.

5.2 Neutral Forms

Careful readers might have noticed that in our grammar of branches, we do not have a case for global variables, nor do we have a β rule for pattern matching on $\texttt{box}\ u^\delta$. So what should $\mathsf{match}\ \texttt{box}\ u^\delta\ \mathsf{with}\ \overrightarrow{b}$ be reduced to? The answer might be surprising: this term in fact is *blocked*. Previously, we mentioned a concern about `isapp` raised by Kavvos [33]. His subsequent analysis concludes that sound intensional operations can only act on globally and locally closed code. This restriction is clearly too strong. After looking into the analysis, we see that this conclusion is based on the assumption that intensional operations reduce on $\texttt{box}\ u^\delta$, which leads to the strong restriction. $\mathsf{match}\ \texttt{box}\ u^\delta\ \mathsf{with}\ \overrightarrow{b}$ should not reduce, just for the same reason $\mathsf{match}\ x\ \mathsf{with}\ \overrightarrow{b}$ should not. They are both waiting for substitutions to supply an actual code to unblock the evaluation. Their only difference is that they act on different substitutions. This observation leads to a renewed definition of neutral forms:

$$\begin{aligned} v &:= \cdots \mid \mathsf{match}\ v\ \mathsf{with}\ \overrightarrow{r} \mid \mathsf{match}\ \texttt{box}\ u^\delta\ \mathsf{with}\ \overrightarrow{r} && \text{(Neutral form (}\mathsf{Ne}\text{))} \\ r &:= \mathsf{var}_x \Rightarrow w \mid \mathsf{zero} \Rightarrow w \mid \mathsf{succ}\ ?u \Rightarrow w \mid \lambda x.?u \Rightarrow w \mid ?u\ ?u' \Rightarrow w \\ &\quad \mid \mathsf{rec}_T\ ?u\ (x\ y.?u')\ ?u'' \Rightarrow w && \text{(Normal branches)} \end{aligned}$$

The definition of normal forms, described by w, remains the same. To obtain $\beta\eta$ normal forms, all branches should be normalized. If u is a scrutinee of a match, its local substitution stays as is because it is considered as code. This adjustment is subtle but critical to give a sound presheaf model.

Moreover, notice that $\texttt{letbox}\ u \leftarrow \texttt{box}\ u'^\delta\ \texttt{in}\ t$ does not get blocked. This difference in computational behaviors is due to different purposes of two elimination forms. `letbox` is primary for code composition and the execution of code, while pattern matching focuses on intensional analysis of code. For this reason, we include both `letbox` and pattern matching as elimination forms. They coexist perfectly at layer 1 because our core theory at layer 0 is unaffected. Without layering, both `letbox` and pattern matching are available everywhere, including inside of `box`, which causes all sorts of complex interactions and makes the computational behavior of the whole type theory difficult to control. This is why former systems based on [17] are so difficult to extend with intensional analysis in a controlled manner. Now we have introduced pattern matching on code and achieved feature ③ outlined in Sec. 1. In the rest of this section, we fix the normalization proof and justify that this system is a proper type theory.

5.3 Adjusting the Presheaf Model

Since we only add an elimination form, we simply extend the model in Sec. 4. The adjustments are shown in Fig. 7. Two additional functions are defined: first, the match function dispatches evaluation to the proper branch based on the input code and evaluates the body with a global substitution and an evaluation environment; second, nfbranch normalizes the body of a branch and obtains a normal branch. Applying nfbranch to $\overrightarrow{b}$ normalizes all branches in $\overrightarrow{b}$. nfbranch is invoked when we normalize a pattern matching on some neutral code.

Let us consider the match case. We first evaluate the scrutinee. If the result is a neutral term, then we simply invoke nfbranch to normalize all branch bodies, and then use reflection to obtain a $[\![T]\!]_{\Psi;\Gamma}$. Each case in nfbranch proceeds similarly. The evaluation of the body continues with the global substitution extended with pattern variables. The evaluated body is then reified to a normal form and thus the resulting branch is also normal. If the result is `box` s, then we match the code s accordingly with a branch and evaluate the body. This is done by calling the match function. Based on the shape of s, the match function looks up $\overrightarrow{b}$ and extends σ accordingly before evaluating the body. For example, if s is a λ, then the branch $\lambda x.?u \Rightarrow t$ is picked, and t is evaluated. The lookup of $\overrightarrow{b}$ must succeed because our pattern matching is covering. However, if s is just a global variable, based on the previous discussion on neutral forms, we must block the evaluation and only normalize the branches. The case forms a neutral form and is essentially the same as if the scrutinee is evaluated to a neutral.

The presheaf interpretation gives a semantic explanation of how layering enables sound pattern matching on code and why it is difficult in purely homogeneous systems. A term of type $\square(\Gamma \vdash T)$ has two different uses: it either stands for code that will be run (due to `letbox`) or it stands for code that will be analyzed (due to pattern matching). In the former case, it is evaluated to a natural transformation in the semantics while in the latter, only its syntactic information is needed. Moreover, these two uses are not mutually exclusive. A program might use both semantic and syntactic information of the same code. To support pattern matching on code, we must maintain both semantic and syntactic information and therefore the evaluation of code must be postponed. In our setting, this evaluation only happens when we evaluate a global variable at layer 1. The evaluation function evaluates the code represented by the global variable and maintains its well-foundedness by decreasing the layer from 1 to 0. Meanwhile, in a homogeneous system without layers, it is no longer clear how to give a well-founded evaluation of a global variable due to the lack of proper measure if intensional analysis is supported.

As we will see very shortly, the intuition above based on two different uses of code must be formalized in the gluing model in order to establish a soundness proof, giving a formal account for the importance of layering.

5.4 Soundness

Recall that in Sec. 4.3, we need a 2-layered model, where the layer-1 model refers to the semantic judgment at layer 0 to support `letbox`. The semantic judgment at layer 0 is defined as a universal quantification over global weakenings and the gluing between local substitutions and evaluation environments:

$$\Psi; \Gamma \Vdash_0 t : T := \forall\ \gamma : \Phi \Longrightarrow_g \Psi \text{ and } \delta \sim \rho \in (\!|\Gamma|\!)^0_{\Phi;\Delta} \,.\, t[\gamma][\delta] \sim [\![t]\!]^0_{\Phi;\Delta}(\gamma;\rho) \in (\!|T|\!)^0_{\Phi;\Delta}$$

$$\llbracket \_ \rrbracket^i_{\Psi;\Gamma} : \Phi;\Delta \vdash_i t : T \to \llbracket \Phi;\Delta \rrbracket^i_{\Psi;\Gamma} \to \llbracket T \rrbracket_{\Psi;\Gamma} \quad \textbf{(Evaluation)}$$

$$\llbracket \mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b} \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) := \mathsf{match}(s, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) \qquad (\text{if } \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) = \mathtt{box}\ s)$$

$$\llbracket \mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b} \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) := \uparrow^T_{\Psi;\Gamma} (\mathsf{match}\ v\ \mathsf{with}\ \overrightarrow{r})$$

$$(\text{if } \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) = v : \Box(\Delta' \vdash S) \text{ for some } \Delta' \text{ and } \overrightarrow{r} := \mathsf{nfbranch}(\overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho))$$

$$\mathsf{match} : \Psi;\Gamma \vdash_0 t : T \to \Phi;\Delta \vdash_1 \overrightarrow{b} : \Delta' \vdash T \Rightarrow T' \to \llbracket \Phi;\Delta \rrbracket^1_{\Psi;\Gamma} \to \llbracket T \rrbracket_{\Psi;\Gamma}$$

$$\textbf{(Branch Evaluation Based on Code)}$$

$$\mathsf{match}(x, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) \qquad (\text{where } x \Rightarrow t := \overrightarrow{b}(x))$$

$$\mathsf{match}(\mathsf{zero}, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) \qquad (\text{where } \mathsf{zero} \Rightarrow t := \overrightarrow{b}(\mathsf{zero}))$$

$$\mathsf{match}(\mathsf{succ}\ s, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma, s/u;\rho) \qquad (\text{where } \mathsf{succ}\ ?u \Rightarrow t := \overrightarrow{b}(\mathsf{succ}\ s))$$

$$\mathsf{match}(\lambda x.s, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma, s/u;\rho) \qquad (\text{where } \lambda x.?u \Rightarrow t := \overrightarrow{b}(\lambda x.s))$$

$$\mathsf{match}(t'\ s, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma, t'/u, s/u;\rho) \qquad (\text{where } ?u\ ?u' \Rightarrow t := \overrightarrow{b}(t'\ s))$$

$$\mathsf{match}(u^\delta, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \uparrow^{T'}_{\Psi;\Gamma} (\mathsf{match}\ \mathtt{box}\ u^\delta\ \mathsf{with}\ \overrightarrow{r})$$

$$(\text{where } \overrightarrow{r} := \mathsf{nfbranch}(\overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho))$$

$$\mathsf{nfbranch} : \Phi;\Delta \vdash_1 b : \Delta' \vdash T \Rightarrow T' \to \llbracket \Phi;\Delta \rrbracket^1_{\Psi;\Gamma} \to \Psi;\Gamma \vdash_1 r : \Delta' \vdash T \Rightarrow T'$$

$$\textbf{(Normalization of A Branch)}$$

$$\mathsf{nfbranch}(x \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) := x \Rightarrow \downarrow^{T'}_{\Psi;\Gamma} (\llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho))$$

$$\mathsf{nfbranch}(\mathsf{zero} \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) := \mathsf{zero} \Rightarrow \downarrow^{T'}_{\Psi;\Gamma} (\llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho))$$

$$\mathsf{nfbranch}(\mathsf{succ}\ ?u \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) :=$$

$$\mathsf{succ}\ ?u \Rightarrow \downarrow^{T'}_{\Psi,u:(\Delta' \vdash \mathtt{Nat});\Gamma} (\llbracket t \rrbracket^1_{\Psi,u:(\Delta' \vdash \mathtt{Nat});\Gamma}(\sigma', u^{\mathsf{id}}/u;\rho'))$$

$$(\text{where } (\sigma';\rho') := (\sigma;\rho)[p(\mathsf{id});\mathsf{id}] \in \llbracket \Phi;\Delta \rrbracket^1_{\Psi,u:(\Delta' \vdash \mathtt{Nat});\Gamma})$$

$$\mathsf{nfbranch}(\lambda x.?u \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) :=$$

$$\lambda x.?u \Rightarrow \downarrow^{T'}_{\Psi,u:(\Delta',x:S \vdash T);\Gamma} (\llbracket t \rrbracket^1_{\Psi,u:(\Delta',x:S \vdash T);\Gamma}(\sigma', u^{\mathsf{id}}/u;\rho'))$$

$$(\text{where } x : S \text{ and } (\sigma';\rho') := (\sigma;\rho)[p(\mathsf{id});\mathsf{id}] \in \llbracket \Phi;\Delta \rrbracket^1_{\Psi,u:(\Delta',x:S \vdash T);\Gamma})$$

$$\mathsf{nfbranch}(?u\ ?u' \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) := ?u\ ?u' \Rightarrow \downarrow^{T'}_{\Psi';\Gamma} (\llbracket t \rrbracket^1_{\Psi';\Gamma}(\sigma', u^{\mathsf{id}}/u, u'^{\mathsf{id}}/u';\rho'))$$

$$(\text{for any } S\ \mathtt{wf}^0, \text{ where } \Psi' := \Psi, u : (\Delta' \vdash S \longrightarrow T'), u' : (\Delta' \vdash S),)$$

$$(\text{and } (\sigma';\rho') := (\sigma;\rho)[p(p(\mathsf{id}));\mathsf{id}] \in \llbracket \Phi;\Delta \rrbracket^1_{\Psi';\Gamma})$$

Fig. 7: Adjustments to the presheaf model

This definition taken from Sec. 4.3, unfortunately, cannot support the semantic rule for pattern matching. Consider some $\Psi;\Gamma \Vdash_0 t\ s : T$. To prove that pattern matching on code is semantically sound, we perform an analysis on the structure of $t\ s$. But we are stuck, because we cannot derive $\Psi;\Gamma \Vdash_0 t : S \longrightarrow T$ or $\Psi;\Gamma \Vdash_0 s : S$ for some S. In general, the semantic information of subterms is lost. To support pattern matching on code, our semantic judgment must maintain both the syntactic structure of the code and the semantic information of all subterms. Therefore, our semantic judgments at layer 0 become inductively defined (Fig. 8). These rules are essentially just the typing rules with some extra $\Psi;\Gamma \Vdash_0 t : T$ premises. The inductive definition makes sure that the semantic information for all subterms are maintained.

Finally, we refer to $\Psi;\Gamma \Vdash_0 t : T$ when we define the gluing relation for $\Box(\Gamma \vdash T)$ at layer 1. This allows us to inspect the syntactic structure of t during an evaluation of pattern matching and access its semantic information at the same time. We refer readers to our technical report [27] for the proofs of the semantic rules for pattern matching. At layer 1, we use the new semantic

$$\frac{\Psi\ \mathtt{wf}^0 \qquad \Gamma\ \mathtt{wf}^0}{\Psi; \Gamma \Vdash_0 \cdot : \cdot} \qquad \frac{\Psi; \Gamma \Vdash_0 \delta : \Delta \qquad \Psi; \Gamma \Vdash_0 t : T}{\Psi; \Gamma \Vdash_0 \delta, t/x : \Delta, x : T} \qquad \frac{\Psi\ \mathtt{wf}^0 \qquad \Gamma\ \mathtt{wf}^0}{\Psi; \Gamma \Vdash_0 \mathsf{zero} : \mathtt{Nat}}$$

$$\frac{\begin{array}{c}\Psi; \Gamma \Vdash_0 t : \mathtt{Nat} \\ \Psi; \Gamma \VDash_0 \mathsf{succ}\ t : \mathtt{Nat}\end{array}}{\Psi; \Gamma \Vdash_0 \mathsf{succ}\ t : \mathtt{Nat}} \qquad \frac{\begin{array}{c}\Psi\ \mathtt{wf}^0 \quad \Gamma\ \mathtt{wf}^i \\ x : T \in \Gamma\end{array}}{\Psi; \Gamma \Vdash_0 x : T} \qquad \frac{\begin{array}{c}\Psi; \Gamma \Vdash_0 \delta : \Delta \\ u : (\Delta \vdash T) \in \Psi \quad \Psi; \Gamma \VDash_0 u^\delta : \Box(\Delta \vdash T)\end{array}}{\Psi; \Gamma \Vdash_0 u^\delta : \Box(\Delta \vdash T)}$$

$$\frac{\begin{array}{c}\Psi; \Gamma, x : S \Vdash_0 t : T \\ \Psi; \Gamma \VDash_0 \lambda x.t : S \longrightarrow T\end{array}}{\Psi; \Gamma \Vdash_0 \lambda x.t : S \longrightarrow T} \qquad \frac{\begin{array}{c}\Psi; \Gamma \Vdash_0 t : S \longrightarrow T \\ \Psi; \Gamma \Vdash_0 s : S \qquad \Psi; \Gamma \VDash_0 t\ s : T\end{array}}{\Psi; \Gamma \Vdash_0 t\ s : T}$$

Fig. 8: Layer-0 semantic judgment

judgment $\Psi; \Gamma \Vdash_0 t : T$ at layer 0 to define the semantic judgment for global substitutions $\Phi \Vdash \sigma : \Psi$, and then define the semantic judgment for terms and local substitutions:

$$\begin{aligned}\Psi; \Gamma \Vdash_1 t : T &:= \forall\ \Phi; \Gamma \Vdash_0 \sigma : \Psi \text{ and } \delta \sim \rho \in (\!|\Gamma|\!)^1_{\Phi;\Delta} . \\ &\qquad\qquad t[\sigma; \delta] \sim [\![t]\!]^1_{\Phi;\Delta}(\sigma; \rho) \in (\!|T|\!)^1_{\Phi;\Delta} \\ \Psi; \Gamma \Vdash_1 \delta' : \Delta' &:= \forall\ \Phi; \Gamma \Vdash_0 \sigma : \Psi \text{ and } \delta \sim \rho \in (\!|\Gamma|\!)^1_{\Phi;\Delta} . \\ &\qquad\qquad \delta'[\sigma] \circ \delta \sim [\![\delta']\!]^1_{\Phi;\Delta}(\sigma; \rho) \in (\!|\Delta'|\!)^1_{\Phi;\Delta}\end{aligned}$$

By proving and then instantiating the fundamental theorems, we obtain the soundness proof.

6 Future Extensions to Layered Systems

We have shown that 2-layered modal type theory supports intensional analysis and retains normalization. In this section, we build on our previous development and describe three possible extensions of layered systems as future work.

6.1 Extending to Complex Type Systems

In this paper, so far we only focused on simple types. Layering, however, is a powerful idea that, we believe, scales naturally. A natural extension is to consider System F, the foundation for many practical programming languages like Haskell and ML family, as the core language. Haskell and ML communities have expressed strong interest in meta-programming [49,59,58,34,35,53, etc.]. Layering provides a simple solution to this problem. In 2-layered System F, we replace validity of types with well-kindedness of types: $\Psi; \Gamma \vdash_i T : *$. Following the matryoshka principle, at layer 0, we operate in System F, while at layer 1, we work in a meta-language extending System F with one layer of $\Box$. We hope that 2-layered System F not only guarantees the well-scopedness and well-typedness of code, but is also normalizing, following our development here.

Besides System F, we are also interested in using Martin-Löf type theory (MLTT) as the base language. 2-layered MLTT would provide a foundation for tactic languages and meta-programming in proof assistants like Coq, Agda and

Lean. Following our previous development, we expect that 2-layered MLTT enables ① the reuse of *all* definitions from layer 0 at layer 1, and ② the guarantee of well-scopedness and well-typedness of all code. Since in MLTT types are also terms, we simply reuse contextual types $\square(\Gamma \vdash \texttt{Type})$ for code of types.

One challenge we expect from extending MLTT with layering is the semantics of code. For example, when we pattern match on code $t : (\lambda x.x)\ \texttt{Nat}$, we expect that the type is reduced to $\texttt{Nat}$. That is, $(\lambda x.x)\ \texttt{Nat}$ and $\texttt{Nat}$ as types are considered the same even for code. We effectively take quotient of code over types. This behavior aligns well with quotient inductive-inductive types (QIIT) [31,6] and we expect QIIT to appear in the semantics in some form, but we leave the detailed investigation as future work.

6.2 Extending Power of Layer 1

Though pattern matching allows inspection of code, not all operations can be defined easily in this way. For example, the weak head normal form reduction (whnf) on a term is not defined by a simple structural recursion on the syntax of the term. In 2-layered modal type theory, we can extend a whnf operation at layer 1 and still maintain normalization. The following are the rewrite rules for whnf and we can extend our previous normalization algorithm in Sec. 5 with a rewrite process [9,10]:

$$\mathsf{whnf}\ (\mathsf{box\ zero}) \rightsquigarrow \mathsf{box\ zero} \quad \mathsf{whnf}\ (\mathsf{box}\ (\mathsf{succ}\ t)) \rightsquigarrow \mathsf{box}\ (\mathsf{succ}\ t)$$
$$\mathsf{whnf}\ (\mathsf{box}\ (\lambda x.t)) \rightsquigarrow \mathsf{box}\ (\lambda x.t) \quad \mathsf{whnf}\ (\mathsf{box}\ ((\lambda x.t)\ s)) \rightsquigarrow \mathsf{whnf}\ (\mathsf{box}\ (t[s/x]))$$
$$\mathsf{whnf}\ (\mathsf{box}\ x) \rightsquigarrow \mathsf{box}\ x \qquad \frac{\mathsf{whnf}\ (\mathsf{box}\ t) \rightsquigarrow \mathsf{whnf}\ (\mathsf{box}\ t')}{\mathsf{whnf}\ (\mathsf{box}\ (t\ s)) \rightsquigarrow \mathsf{whnf}\ (\mathsf{box}\ (t'\ s))}$$

whnf does not go under succ or λ and is only available at layer 1. Both local and global substitutions simply propagate under whnf. The rewrite process repeats these rules until no rule matches. This process will terminate due to the strong normalization of STLC and therefore the whole system remains terminating. However, with global variables, we must apply extra care to maintain confluence and eventually normalization. In Sec. 5.2, we discuss the impact of global substitutions and the necessity of their stability. When we extend layer 1 with another operation, we must also make sure that this extended operation is stable under global substitutions. When whnf encounters a global variable in the head position, such as $\mathsf{whnf}(\mathsf{box}\ u^\delta)$ or $\mathsf{whnf}(\mathsf{box}\ (u^\delta\ s))$, there is no matching rule and the rewrite process stops for the same reason for pattern matching stopping for $\mathsf{box}\ u^\delta$. The lack of a reduction rule when a global variable is in the head position is particularly important. With whnf, we can now simplify a term before matching, which is a very useful and typical tactic in proof assistants:

$$\mathsf{match\ box}\ ((\lambda x.x)\ \mathsf{zero})\ \mathsf{with} \qquad \mid \mathsf{zero} \Rightarrow \mathsf{true} \mid \_ \Rightarrow \mathsf{false} \approx \mathsf{false}$$
$$\mathsf{match\ whnf}\ (\mathsf{box}\ ((\lambda x.x)\ \mathsf{zero}))\ \mathsf{with} \mid \mathsf{zero} \Rightarrow \mathsf{true} \mid \_ \Rightarrow \mathsf{false} \approx \mathsf{true}$$

Due to layering, whnf only needs to consider terms in STLC at layer 0. In a homogeneous system, whnf must apply to all possible code, and thus becomes troublesome to define. This extensibility of layer 1 is an important and useful feature for a foundation for meta-programming in proof assistants.

6.3 Extending to More Layers

Another potential of layering is to generalize the 2-layered system to n layers for a fixed $n > 2$. Scaling to n layers is in fact technically detailed, but conceptually simple. We sketch the process briefly here and leave the details as future work.

In a layered system, terms are type-checked in a *context array*. For an n-layered system, this context array has length n:

$$\Gamma_{n-1}; \cdots ; \Gamma_1; \Gamma_0 \vdash_i t : T \qquad \text{or} \qquad \vec{\Gamma} \vdash_i t : T \qquad \text{where } i \in [0, n-1].$$

We now use x, y to range over variables in all contexts. Each context in the context array contains bindings of a fixed shape. Bindings in Γ_0 are $x : T$. Bindings in Γ_i for $i \in [1, n-1]$ are of the shape $x : (\Delta_{i-1}; \cdots ; \Delta_0 \vdash T)$. Bindings in each Δ_j also have the specified shape. Contextual types are generalized to context arrays: $\square(\Delta_{i-1}; \cdots ; \Delta_0 \vdash T)$. The design of a n-layered system is guided by two principles: the matryoshka principle, which says types and terms at lower layers are subsumed by higher layers, and the static code principle, which only terms at layer $n-1$ compute. Particularly, the latter principle means that terms from layer 0 to $n-2$ are static code. Following both principles, we will be able to fill in the details and design an n-layered system.

We expect the n-layered generalization to be compatible with the extension with operations described in Sec. 6.2. Instead of extending layer 1, we extend layer $n-1$ so that all lower layers are unaffected.

7 Related Work and Conclusion

7.1 Normalization of Modal Type Theories

The core of our paper is the normalization of 2-layered modal type theory. Recently, there have been a number of approaches that explore modal type theories. One of the earliest is from Nanevski et al. [39]. They prove the normalization for contextual modal type theory (CMTT) indirectly by a translation to the system by de Groote [23]. This translation does not give a direct normalization algorithm for CMTT. Our system in Sec. 3 is strictly weaker than CMTT by disallowing nested $\square$s. Even if we scale our system to n layers as outlined in Sec. 6.3, the resulting system will only have a hierarchy of contextual types, so we still cannot recover the same expressive power as CMTT to do arbitrary nesting of $\square$s. Nevertheless, in Sec. 5, we have shown that this temporary loss in expressive power enables an orthogonal avenue of intensional analysis that is difficult to obtain in CMTT. Kavvos [32] proposes formulations for a few different modalities in the dual-context style and proves the normalization using a translation to de Groote's system as well. The normalization of System GL, however, is proved directly by reducibility candidates. Lately, Gratzer [20] proves the normalization of multimodal type theory, a generalization of the dual-context systems, using Sterling's [51] synthetic Tait's computability.

The Kripke-style systems are another kind of formulation of modalities and is different from ours in context management. The normalization problem for this style is more intensively investigated. Borghuis [13] proves the strong normalization of modal pure type systems in his PhD thesis. More recently, Clouston [16] proves the normalization of System K using reducbility candidates. Gratzer et al.[22] prove the normalization of a dependent type theory with idempotent $S4$ by parameterizing Abel's [1] untyped domain method with a poset. Valliappan et al. [55] use the same method and prove the normalization for Systems K, T, $K4$ and $S4$. Hu and Pientka [26] establish the same result but introduce a "truncoid" algebraic structure to the untyped domain model instead, so that one normalization proof can be instantiated to adapt to all four systems. This method using truncoids has been scaled to dependent types [25]. It is worth emphasizing that none of these modal type theories supports pattern matching on code as we do in our 2-layer modal type theory.

7.2 Homogeneous Meta-programming and Its Foundations

Early ideas of metaprogramming using quasi-quoting style can be traced back to Lisp/Scheme [3]. In Lisp's untyped setting, all programs are represented as lists, so intensional analysis is reduced to inspections of lists and is relatively simple. Supporting type-safe metaprogramming leads to all the complications. MetaML [54] is an early example for type-safe meta-programming. MetaML employs a quasi-quoting style similar to Lisp. However, MetaML does not support any form of intensional analysis. In fact, MetaML's meta-theory even allows reduction of code under quote [52], so intensional analysis is deliberately avoided. The correspondence between meta-programming and modal logic S4 is described by Davies and Pfenning [17]. The correspondence explains how the modal logic $S4$ models multi-staging and code composition, but it does not explain how intensional analysis should be supported. Two formulations of $S4$ are presented: the dual-context style and the Kripke style. While the Kripke-style formulation provides a type-theoretic formulation for quasi-quotation, it is more challenging to extend and support intensional analysis. On the other hand, the dual-context style forces programmers to write meta-programs in a comonadic style, but it has better setup for intensional analysis as we have demonstrated. This is also the approach taken in Beluga [43,45] and Moebius [30].

The semantics for dual-context style has also been studied previously. In the context of contextual types, Gabbay and Nanevski [19] attempt to give a set-theoretic semantics to contextual types. As pointed out by Kavvos [33], their exact formulation of contextual types seems to break the confluence property.

Boespflug and Pientka [12] extend the dual-context style to the multi-context style. Though the multi-context style and the Kripke style both use multiple contexts for typing, the number of contexts in the former is more or less fixed (hence context arrays), while in the latter, contexts are often pushed and popped during typing (hence context stacks). Davies and Pfenning [17] show that the

Kripke style system is equivalent to the dual-context style. Moebius [30] combines the multi-context style and contextual types, and supports pattern matching on code in System F. Moebius has subject reduction. However, to adapt Moebius to a type theory, normalization must be proved, but it is not obvious how to support coverage. Whether layering provides a solution requires a future investigation.

*Ω*mega [56] is another example for a sound meta-programming system with pattern matching on code. *Ω*mega implements the quasi-quoting style. The open context of a code is annotated in the type, similar to contextual types. However, the type of the code itself is not remembered, so their type system is not as complex due to reduced type information.

7.3 Intensionality in Type Theories

Interest in intensionality is often associated with modalities. Pfenning [41] describes a type theory in which terms might be treated intensionally, extensionally, or irrelevantly when corresponding modalities are employed. This is similar to our setting, where intensionality of code is marked by the $\square$ modality. In the same setting, Kavvos [33] describes a special kind of intensional recursions using Löb induction ($(\square A \to A) \to A$), which supports meta-programs to access their own code. Löb induction says if we can prove A from the proof of A, then A is true. Löb induction is incompatible with the Axiom T ($\square A \to A$), but it still has interesting computational behaviors, including an example for computer viruses. Chen and Ko [14] resolve the incompatibility between the Löb induction and the Axiom T by supporting them in two separate modalities.

7.4 Layering in Type Theories

Layering can also be found in other type theories. Logical Framework (LF) [24] is essentially a layered system. LF is a dependently typed framework to define object languages. These object languages live at one layer. LF as their meta-language live at a higher layer. Isabelle [40] is one example for modern proof assistants based on LF. There are two layers in Cocon [47]. At the lower layer is LF, which defines an object language. At the higher layer is a Martin-Löf type theory (MLTT) for computation. Two layers are connected by contextual types. Cocon supports induction in MLTT on the syntax of an object language in LF. Cocon's structure leads to a similar semantics to our 2-layered modal type theory. The main difference is that in 2-layered modal type theory, the core language at layer 0 is a sub-language of the computational language at layer 1. Consequently, all terms at layer 0 can be lifted to layer 1 (Lemma 3) for free and be run as programs. The conversion from code to programs is done implicitly in the semantics. Contrarily, in Cocon, since the object language is defined freely, an embedding to MLTT must be given explicitly and is only possible if the object language has strictly weaker expressive power. A categorical semantics for Cocon is given by Hu et al.[46,29]. Kovács [36] and Allais [4] demonstrate applications of 2-level type theory which focuses more on code composition and does not support intensional analysis.

Our system uses layers to account for the number of nested $\square$'s, which shares some similarity with graded and quantitative systems [8,2,38]. The latter systems use grades to keep track of uses of variables. We believe that it would be interesting to have a universal framework to contain all these different uses of modalities, though it requires further investigations.

Our approach is also similar to GuTT [21], a guarded type theory supporting Löb induction. GuTT has two layers. The first layer excludes dynamics of Löb induction (but not for other terms) and enjoys normalization. The lost dynamics is recovered at the second layer, at the cost of normalization. We are similar in that we both take advantage of differences between layers and one layer is the extension of the other.

7.5 Conclusion and Future Work

In this paper, we introduce the layered style to support intensional analysis in type theory. In the layered view, meta-programming is done in an extended language of a chosen core language. Pattern matching on code at a higher layer only needs to handle code at lower layers, hence circumventing the complications in previous work. We investigate the layered style in 2-layered modal type theory, which supports pattern matching on code where we guarantee coverage by construction. We provide a constructive proof of normalization by evaluation using a presheaf model. The normalization algorithm extracted from the model is proven complete and sound and is implemented in Agda.

Layering provides a controlled and modular way to introduce meta-programming with intensional analysis to type theory. As a first step, we plan to add context abstraction following the approach taken for example in Beluga to support more general recursion principles [43,44]. We see abstracting over contexts as an orthogonal issue. As a next step, we will adapt layering to Martin-Löf type theory (MLTT). Both extensions will create the first dependent type theory that is supporting intensional analysis of code within MLTT. In the long term, we hope that this type theory will also provide a foundation for extending the core language of Coq, Agda, or other proof assistants with a meta-language of tactics that can reuse *all* definitions from the core language while the normalization of the overall system is retained.

References

1. Abel, A.: Normalization by evaluation: dependent types and impredicativity. Habilitation thesis, Ludwig-Maximilians-Universität München (2013), https://www.cse.chalmers.se/~abela/habil.pdf
2. Abel, A., Bernardy, J.: A unified view of modalities in type systems. Proc. ACM Program. Lang. **4**(ICFP), 90:1–90:28 (2020), https://doi.org/10.1145/3408972
3. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs, Second Edition. MIT Press (1996)
4. Allais, G.: Scoped and typed staging by evaluation (2024), https://arxiv.org/abs/2310.13413

5. Altenkirch, T., Hofmann, M., Streicher, T.: Categorical reconstruction of a reduction free normalization proof. In: Pitt, D.H., Rydeheard, D.E., Johnstone, P.T. (eds.) Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings. Lecture Notes in Computer Science, vol. 953, pp. 182–199. Springer (1995), https://doi.org/10.1007/3-540-60164-3_27
6. Altenkirch, T., Kaposi, A.: Normalisation by evaluation for dependent types. In: Kesner, D., Pientka, B. (eds.) 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal. LIPIcs, vol. 52, pp. 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016), https://doi.org/10.4230/LIPIcs.FSCD.2016.6
7. Anand, A., Boulier, S., Cohen, C., Sozeau, M., Tabareau, N.: Towards certified meta-programming with typed template-coq. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10895, pp. 20–39. Springer (2018), https://doi.org/10.1007/978-3-319-94821-8_2
8. Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65. ACM (2018), https://doi.org/10.1145/3209108.3209189
9. Berger, U., Eberl, M., Schwichtenberg, H.: Normalization by Evaluation. In: Möller, B., Tucker, J.V. (eds.) Prospects for Hardware Foundations: ESPRIT Working Group 8533 NADA — New Hardware Design Methods Survey Chapters, pp. 117–137. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (1998), https://doi.org/10.1007/3-540-49254-2_4
10. Berger, U., Eberl, M., Schwichtenberg, H.: Term rewriting for normalization by evaluation. Inf. Comput. **183**(1), 19–42 (2003), https://doi.org/10.1016/S0890-5401(03)00014-2
11. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed lambda -calculus. In: [1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science. pp. 203–211 (Jul 1991). https://doi.org/10.1109/LICS.1991.151645
12. Boespflug, M., Pientka, B.: Multi-level contextual type theory. In: Geuvers, H., Nadathur, G. (eds.) Proceedings Sixth International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2011, Nijmegen, The Netherlands, August 26, 2011. EPTCS, vol. 71, pp. 29–43 (2011), https://doi.org/10.4204/EPTCS.71.3
13. Borghuis, V.A.J.: Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus. PhD Thesis, Mathematics and Computer Science (1994), https://doi.org/10.6100/IR427575
14. Chen, L., Ko, H.: Realising intensional S4 and GL modalities. In: Manea, F., Simpson, A. (eds.) 30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference). LIPIcs, vol. 216, pp. 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), https://doi.org/10.4230/LIPIcs.CSL.2022.14
15. Christiansen, D.R., Brady, E.C.: Elaborator reflection: extending idris in idris. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan,

September 18-22, 2016. pp. 284–297. ACM (2016), https://doi.org/10.1145/2951913.2951932

16. Clouston, R.: Fitch-style modal lambda calculi. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10803, pp. 258–275. Springer (2018), https://doi.org/10.1007/978-3-319-89366-2_14
17. Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM **48**(3), 555–604 (May 2001), https://dl.acm.org/doi/10.1145/382780.382785
18. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proc. ACM Program. Lang. **1**(ICFP), 34:1–34:29 (2017), https://doi.org/10.1145/3110278
19. Gabbay, M.J., Nanevski, A.: Denotation of contextual modal type theory (CMTT): syntax and meta-programming. J. Appl. Log. **11**(1), 1–29 (2013), https://doi.org/10.1016/j.jal.2012.07.002
20. Gratzer, D.: Normalization for multimodal type theory. In: Baier, C., Fisman, D. (eds.) LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022. pp. 2:1–2:13. ACM (2022), https://doi.org/10.1145/3531130.3532398
21. Gratzer, D., Birkedal, L.: A stratified approach to löb induction. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel. LIPIcs, vol. 228, pp. 23:1–23:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), https://doi.org/10.4230/LIPIcs.FSCD.2022.23
22. Gratzer, D., Sterling, J., Birkedal, L.: Implementing a modal dependent type theory. Proceedings of the ACM on Programming Languages **3**(ICFP), 107:1–107:29 (Jul 2019), https://doi.org/10.1145/3341711
23. de Groote, P.: On the Strong Normalization of Natural Deduction with Permutation-Conversions. In: Narendran, P., Rusinowitch, M. (eds.) Rewriting Techniques and Applications. pp. 45–59. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (1999), https://doi.org/10.1007/3-540-48685-2_4
24. Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. J. ACM **40**(1), 143–184 (1993), https://doi.org/10.1145/138027.138060
25. Hu, J.Z.S., Jang, J., Pientka, B.: Normalization by evaluation for modal dependent type theory. Journal of Functional Programming **33**, e7 (2023). https://doi.org/10.1017/S0956796823000060
26. Hu, J.Z.S., Pientka, B.: A Categorical Normalization Proof for the Modal Lambda-Calculus. Electronic Notes in Theoretical Informatics and Computer Science **Volume 1 - Proceedings of MFPS XXXVIII** (Feb 2023), https://entics.episciences.org/10360
27. Hu, J.Z.S., Pientka, B.: Layered modal type theories (2023), https://arxiv.org/abs/2305.06548
28. Hu, J.Z.S., Pientka, B.: Agda mechanization (2024), https://doi.org/10.5281/zenodo.10492818
29. Hu, J.Z.S., Pientka, B., Schöpp, U.: A category theoretic view of contextual types: From simple types to dependent types. ACM Trans. Comput. Log. **23**(4), 25:1–25:36 (2022), https://doi.org/10.1145/3545115

30. Jang, J., Gélineau, S., Monnier, S., Pientka, B.: Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022), `https://doi.org/10.1145/3498700`
31. Kaposi, A., Altenkirch, T.: Normalisation by Evaluation for Type Theory, in Type Theory. Logical Methods in Computer Science **Volume 13, Issue 4** (Oct 2017), `https://lmcs.episciences.org/4005/pdf`, publisher: Episciences.org
32. Kavvos, G.A.: Dual-context calculi for modal logic. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017. pp. 1–12. IEEE Computer Society (2017), `https://doi.org/10.1109/LICS.2017.8005089`
33. Kavvos, G.A.: Intensionality, intensional recursion and the gödel-löb axiom. FLAP **8**(8), 2287–2312 (2021), `https://collegepublications.co.uk/ifcolog/?00050`
34. Kiselyov, O.: The design and implementation of BER metaocaml - system description. In: Codish, M., Sumii, E. (eds.) Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8475, pp. 86–102. Springer (2014), `https://doi.org/10.1007/978-3-319-07151-0_6`
35. Kokaji, Y., Kameyama, Y.: Polymorphic multi-stage language with control effects. In: Yang, H. (ed.) Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7078, pp. 105–120. Springer (2011), `https://doi.org/10.1007/978-3-642-25318-8_11`
36. Kovács, A.: Staged compilation with two-level type theory. Proc. ACM Program. Lang. **6**(ICFP), 540–569 (2022), `https://doi.org/10.1145/3547641`
37. Martin-Löf, P.: An Intuitionistic Theory of Types: Predicative Part. In: Rose, H.E., Shepherdson, J.C. (eds.) Studies in Logic and the Foundations of Mathematics, Logic Colloquium '73, vol. 80, pp. 73–118. Elsevier (Jan 1975), `https://www.sciencedirect.com/science/article/pii/S0049237X08719451`
38. Moon, B., III, H.E., Orchard, D.: Graded modal dependent type theory. In: Yoshida, N. (ed.) Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12648, pp. 462–490. Springer (2021), `https://doi.org/10.1007/978-3-030-72019-3_17`
39. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic **9**(3), 23:1–23:49 (Jun 2008), `https://doi.org/10.1145/1352582.1352591`
40. Paulson, L.C.: Natural deduction as higher-order resolution. J. Log. Program. **3**(3), 237–258 (1986), `https://doi.org/10.1016/0743-1066(86)90015-4`
41. Pfenning, F.: Intensionality, extensionality, and proof irrelevance in modal type theory. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings. pp. 221–230. IEEE Computer Society (2001), `https://doi.org/10.1109/LICS.2001.932499`
42. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical Structures in Computer Science **11**(04) (Aug 2001), `http://www.journals.cambridge.org/abstract_S0960129501003322`
43. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Pro-

gramming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 371–382. ACM (2008), https://doi.org/10.1145/1328438.1328483
44. Pientka, B., Abel, A.: Well-Founded Recursion over Contextual Objects. In: Altenkirch, T. (ed.) 13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 38, pp. 273–287. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015), http://drops.dagstuhl.de/opus/volltexte/2015/5169
45. Pientka, B., Dunfield, J.: Programming with proofs and explicit contexts. In: Antoy, S., Albert, E. (eds.) Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain. pp. 163–173. ACM (2008), https://doi.org/10.1145/1389449.1389469
46. Pientka, B., Schöpp, U.: Semantical Analysis of Contextual Types. In: Goubault-Larrecq, J., König, B. (eds.) Foundations of Software Science and Computation Structures. pp. 502–521. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-45231-5_26
47. Pientka, B., Thibodeau, D., Abel, A., Ferreira, F., Zucchini, R.: A type theory for defining logics and proofs. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–13. IEEE (2019), https://doi.org/10.1109/LICS.2019.8785683
48. Schürmann, C., Despeyroux, J., Pfenning, F.: Primitive recursion for higher-order abstract syntax. Theor. Comput. Sci. **266**(1-2), 1–57 (Sep 2001), http://dx.doi.org/10.1016/S0304-3975(00)00418-7
49. Sheard, T., Jones, S.P.: Template meta-programming for haskell. In: Chakravarty, M.M.T. (ed.) Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, Pittsburgh, Pennsylvania, USA, October 3, 2002. pp. 1–16. ACM (2002), https://doi.org/10.1145/581690.581691
50. Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., Winterhalter, T.: The metacoq project. J. Autom. Reason. **64**(5), 947–999 (2020), https://doi.org/10.1007/s10817-019-09540-0
51. Sterling, J.: First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory. Ph.D. thesis, Carnegie Mellon University, USA (2022), https://doi.org/10.1184/r1/19632681.v1
52. Taha, W.: A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of metaml is non-trivial (extended abstract). In: Lawall, J.L. (ed.) Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22-23, 2000. pp. 34–43. ACM (2000), https://doi.org/10.1145/328690.328697
53. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 203–217. PEPM '97, Association for Computing Machinery, New York, NY, USA (Dec 1997), https://doi.org/10.1145/258993.259019
54. Taha, W., Sheard, T.: Metaml and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1-2), 211–242 (2000), https://doi.org/10.1016/S0304-3975(00)00053-0
55. Valliappan, N., Ruch, F., Tomé Cortiñas, C.: Normalization for fitch-style modal calculi. Proc. ACM Program. Lang. **6**(ICFP), 772–798 (2022), https://doi.org/10.1145/3547649

56. Viera, M., Pardo, A.: A multi-stage language with intensional analysis. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. pp. 11–20. ACM (2006), https://doi.org/10.1145/1173706.1173709
57. van der Walt, P., Swierstra, W.: Engineering proof by reflection in agda. In: Hinze, R. (ed.) Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8241, pp. 157–173. Springer (2012), https://doi.org/10.1007/978-3-642-41582-1_10
58. Xie, N., Pickering, M., Löh, A., Wu, N., Yallop, J., Wang, M.: Staging with class: a specification for typed template haskell. Proc. ACM Program. Lang. **6**(POPL), 1–30 (2022), https://doi.org/10.1145/3498723
59. Yallop, J.: Staged generic programming. Proc. ACM Program. Lang. **1**(ICFP), 29:1–29:29 (2017), https://doi.org/10.1145/3110273
60. Zeilberger, N.: Focusing and higher-order abstract syntax. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 359–369. ACM (2008), https://doi.org/10.1145/1328438.1328482
61. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: A monad for typed tactic programming in coq. J. Funct. Program. **25** (2015), https://doi.org/10.1017/S0956796815000118

Program Synthesis from Graded Types

Jack Hughes[1] (✉) and Dominic Orchard[1,2]

[1] University of Kent, Canterbury, UK
joh6@kent.ac.uk
[2] University of Cambridge, Cambridge, UK

Abstract. Graded type systems are a class of type system for fine-grained quantitative reasoning about data-flow in programs. Through the use of resource annotations (or *grades*), a programmer can express various program properties at the type level, reducing the number of typeable programs. These additional constraints on types lend themselves naturally to type-directed *program synthesis*, where this information can be exploited to constrain the search space of programs. We present a synthesis algorithm for a graded type system, where grades form an arbitrary pre-ordered semiring. Harnessing this grade information in synthesis is non-trivial, and we explore some of the issues involved in designing and implementing a resource-aware program synthesis tool. In our evaluation we show that by harnessing grades in synthesis, the majority of our benchmark programs (many of which involve recursive functions over recursive ADTs) require less exploration of the synthesis search space than a purely type-driven approach and with fewer needed input-output examples. This *type-and-graded-directed* approach is demonstrated for the research language Granule but we also adapt it for synthesising Haskell programs that use GHC's linear types extension.

1 Introduction

Type-directed program synthesis is a technique for synthesising programs from user-provided type specifications. The technique has a long history intertwined with proof search, thanks to the Curry-Howard correspondence [37, 22]. We present a program synthesis approach that leverages the information of *graded type systems* that track and enforce program properties related to data flow. Our approach follows the concept of program synthesis as a form of proof search in logic: given a type A we want to find a program term t which inhabits A. We express this in terms of a synthesis *judgement* akin to typing or proof rules:

$$\Gamma \vdash A \Rightarrow t$$

meaning that the term t can be synthesised for the goal type A under a context of assumptions Γ. A calculus of synthesis *rules* for inductively defines the above synthesis judgement for each type former of a language. For example, we may define a synthesis rule for standard product types in the following way:

$$\frac{\Gamma \vdash A \Rightarrow t_1 \qquad \Gamma \vdash B \Rightarrow t_2}{\Gamma \vdash A \times B \Rightarrow (t_1, t_2)}\times_{\text{INTRO}}$$

Reading ‘clockwise’ from the bottom-left: to synthesise a value of type $A \times B$, we synthesise a value of type A and then a value of type B and combine them into a pair in the conclusion. The ‘ingredients’ for synthesising the subterms t_1 and t_2 come from the free-variable assumptions Γ and any constructors of A and B.

Depending on the context, there may be many possible combinations of assumptions to synthesise a pair. Consider the following type and partial program with a *hole* (marked ?) specifying a position to perform program synthesis:

$$f : A \to A \to A \to A \times A \qquad f\ x\ y\ z = ?$$

The function has three parameters all of type A which can be used to synthesise an expression of the goal type $A \times A$. Expressing this synthesis problem as an instantiation of the above $\times_{\text{INTRO}}$ rule yields:

$$\dfrac{x : A, y : A, z : A \vdash A \Rightarrow t_1 \qquad x : A, y : A, z : A \vdash A \Rightarrow t_2}{x : A, y : A, z : A \vdash A \times A \Rightarrow (t_1, t_2)}\ \times_{\text{INTRO}}$$

Even in this simple setting, the number of possibilities starts to become unwieldy: there are 3^2 possible candidate programs based on combinations of x, y and z. We thus wish to constrain the number of choices required by the synthesis algorithm. Many systems achieve this by allowing the user to specify additional information about the desired program behaviour. For example, recent work extends type-directed synthesis to refinement types [50], cost specifications [35], differential privacy [52], ownership information [16], example-guided synthesis [15, 2] and examples integrated with types [17, 47]. The general idea is that the proof search / program synthesis procedure can be pruned and refined given more information, whether richer types, additional examples, or behavioural specifications.

We instead leverage the information contained in *graded type systems* which constrain how data can be used by a program and thus reduce the number of possible synthesis choices. Our hypothesis is that grade-and-type-directed synthesis reduces the number of paths that need to be explored and the number of input-output examples that are needed, thus potentially speeding up synthesis.

Graded type systems trace their roots to linear logic. In linear logic, data is treated as though it were a finite resource which must be consumed exactly once, disallowing arbitrary copying and discarding [20]. Non-linearity is captured by the ! modal operator (the *exponential modality*). This gives a binary view—a value may either be used exactly once or in a completely unconstrained way. Bounded Linear Logic (BLL) refines this view, replacing ! with a family of indexed modal operators where the index provides an upper bound on usage [21], e.g., $!_{\leq 4}A$ represents a value A which may be used up to 4 times. Various works have generalised BLL, resulting in *graded* type systems in which these indices are drawn from an arbitrary pre-ordered semiring [12, 19, 49, 1, 14, 5, 39]. This allows numerous program properties to be tracked and enforced statically. Such systems are being deployed in language implementations, forming the basis of Haskell’s linear types extension [8], Idris 2 [11], and the language Granule [45].

Returning to our example in a graded setting, the function’s parameters now have *grades* that we choose, for the sake of example, to be particular natural numbers describing the exact number of times the parameters must be used:

$$f : A^2 \to A^0 \to A^0 \to A \times A \qquad f\ x\ y\ z = ?$$

The first A is annotated with a grade 2 meaning it *must* be used twice. The types of y and z are graded with 0, enforcing zero usage, i.e., they cannot be used in the body of f. The result is that there is only one (normal form) inhabitant for this type: (x, x). For synthesis, the other assumptions will not even be considered, allowing pruning of branches which use resources in a way which violates the grades. Natural number grades in this example explain how many times a value can be used, but we may instead represent different program properties such as sensitivity, strictness, or security levels for tracking non-interference, all of which are well-known instances of graded type systems [45, 18, 1]. These examples are all graded presentations of *coeffects*, tracking how a programs uses its context, in contrast with graded types for *effects* [46, 32] which are not considered here.

In prior work, we built on proof search for linear logic [25], developing a program synthesis technique for a linear type theory with graded modalities $\Box_r A$ (where r is drawn from a semiring) and non-recursive types [27], which we refer to as LGM i.e., *linear-graded-modal*. We adapt some of these ideas to a setting which does not have a linear basis, but rather a type system where grades are pervasive (such as the core of Haskell's linear types extension [8]) alongside recursive algebraic data types and input-output example specifications.

We make the following contributions:

- We define a synthesis calculus for a core graded type system, adapting the context management scheme of LGM to a fully graded setting (rather than the linear setting) and also addressing recursion, recursive types, and user-defined ADTs, none of which were considered in previous work. Synthesised is proved sound, i.e., synthesised programs are typed by the goal type.
- We implemented both the core type system as an extension of Granule and implemented the synthesis calculus algorithmically.[1] We elide full details of the implementation but explain its connection to the formal development.
- We extend the Granule language to include input-output examples as specifications with first-class syntax (that is type checked), which complements the synthesis algorithm and helps guide synthesis. This also aids our evaluation.
- We evaluate our tool on a benchmark suite of recursive functional programs leveraging standard data types like lists, streams, and trees. We compare against non-graded synthesis provided by MYTH [47].
- Leveraging our calculus and implementation, we provide a prototype tool for synthesising Haskell programs that use GHC 9's linear types extension.

Roadmap Section 2 gives a brief overview of proof search in resourceful settings, recalling the 'resource management problem'. Section 3 then defines a core calculus as the target of our synthesis approach. This type system closely resembles various other graded systems [49, 39, 5, 1] including the core of Linear Haskell [8]. We implemented this system as a language extension of Granule [45].

[1] Available at: `github.com/granule-project/granule/releases/tag/v0.9.3.0`

Section 4 presents a calculus of synthesis rules for our language, showing how grades enforce resource usage potentially leading to pruning of the search space of candidate programs. We also discuss some details of the implementation of our tool. We observe the close connection between synthesis in a graded setting and automated theorem proving for linear logic, allowing us to exploit existing optimisation techniques, such as the idea of a focused proof [4].

Section 5 evaluates our implementation on a set of 46 benchmarks, including several non-trivial programs which use algebraic data types and recursion.

Section 6 demonstrates the practicality and versatility of our approach by retargeting our algorithm to synthesise programs in Haskell from type signatures that use GHC's *linear types* extension (which is a graded type system [8]).

2 Overview of Resourceful Program Synthesis

Section 1 discussed synthesising pairs and how graded types could control the number of times assumptions are used in a synthesised term. In a linear or graded setting, synthesis must handle the *resource management problem* [24, 13]: how do we give a resourceful accounting to the context during synthesis, respecting its constraints? We overview the main ideas for addressing this problem.

Section 1 considered (Cartesian) product types $\times$, but we now switch to the *multiplicative product* of linear types, which has the typing rule [20]:

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1, \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\otimes$$

Each subterm is typed by different contexts, which are combined by *disjoint* union: a pair cannot be formed if variables are shared between Γ_1 and Γ_2, preventing the structural behaviour of *contraction* where variables appear in multiple subterms. Naïvely converting this typing rule into a synthesis rule yields:

$$\frac{\Gamma_1 \vdash A \Rightarrow t_1 \qquad \Gamma_2 \vdash B \Rightarrow t_2}{\Gamma_1, \Gamma_2 \vdash A \otimes B \Rightarrow (t_1, t_2)}\otimes\text{INTRO}$$

As a declarative specification, this synthesis rule is sufficient. However, this rule embeds a considerable amount of non-determinism when considered from an algorithmic perspective. Reading 'clockwise' starting from the bottom-left, given a context Γ and a goal $A \otimes B$, we have to split Γ into disjoint subparts Γ_1 and Γ_2 such that $\Gamma = \Gamma_1, \Gamma_2$ in order to pass Γ_1 and Γ_2 to the subgoals for A and B. For a context of size n there are 2^n possible such partitions! This quickly becomes intractable. Instead, Hodas and Miller developed a technique for linear logic programming [25], refined by Cervesato et al. [13], where proof search has an *input context* of available resources and an *output context* of the remaining resources, which we write as judgments $\Gamma \vdash A \Rightarrow^- t \mid \Gamma'$ for input context Γ and output context Γ'. Synthesis for multiplicative products then becomes:

$$\frac{\Gamma_1 \vdash A \Rightarrow^- t_1 \mid \Gamma_2 \qquad \Gamma_2 \vdash B \Rightarrow^- t_2 \mid \Gamma_3}{\Gamma_1 \vdash A \otimes B \Rightarrow^- (t_1, t_2) \mid \Gamma_3}\otimes^-_{\text{INTRO}}$$

The resources remaining after synthesising the term t_1 for A are Γ_2, which are then passed as the resources for synthesising the term of goal type B. There is an ordering implicit here in 'threading through' the contexts between the premises. For example, starting with a context $x : A, y : B$, this rule can be instantiated:

$$\frac{x : A, y : B \vdash A \Rightarrow^{-} x \mid y : B \qquad y : B \vdash B \Rightarrow^{-} y \mid \emptyset}{x : A, y : B \vdash A \otimes B \Rightarrow^{-} (x, y) \mid \emptyset}\ \otimes^{-}_{\text{INTRO}} \qquad \text{(example)}$$

This avoids the problem of splitting the input context, facilitating efficient proof search for linear types. LGM adapted this idea to linear types augmented with graded modalities [27]. We call the above approach *subtractive resource management* due to its similarity to *left-over* type-checking for linear types [3, 54]. In a graded modal setting however this approach is costly [27].

Graded type systems, as considered here, have typing contexts in which free variables are assigned a type and a *grade*: an element of a semiring. For example, the semiring of natural numbers describes how many times an assumption can be used, in contrast to linear assumptions which must be used exactly once, e.g., the context $x :_2 A, y :_0 B$ says that x must be used twice but y cannot be used. The literature contains many example semirings for tracking other properties as graded types, e.g., security labels [18, 1], intervals of usage [45], and hardware schedules [19]. In a graded setting, the subtractive approach is problematic though as there is not necessarily a notion of subtraction for grades.

Consider the above example but for a context with grades r and s on the variables. Using a variable to synthesise a subterm no longer results in that variable being 'left out' of the output context. Instead a new grade is given in the output context relating to the input with a constraint capturing the usage:

$$\frac{\begin{array}{ll}\exists r'.r' + 1 = r & x :_r A, y :_s B \vdash A \Rightarrow^{-} x \mid x :_{r'} A, y :_s B \\ \exists s'.s' + 1 = s & x :_{r'} A, y :_s B \vdash B \Rightarrow^{-} y \mid x :_{r'} A, y :_{s'} B\end{array}}{x :_r A, y :_s B \vdash A \otimes B \Rightarrow^{-} (x, y) \mid x :_{r'} A, y :_{s'} B}\ \otimes^{-}_{\text{INTRO}} \qquad \text{(example)}$$

In the first premise, x has grade r in the input context and x is synthesised for the goal, thus the output context has some grade r' where $r' + 1 = r$, denoting a use of x by the 1 element of the semiring. The second premise is similar.

For the natural numbers, if $r = s = 1$ then the above constraints are satisfied by $r' = s' = 0$. In general, subtractive synthesis for graded types requires solving many such existential equations over semirings, which introduces a new source of non-determinism as there can be more than one solution. LGM implemented this approach, leveraging SMT solving in the context of the Granule language, but show that a dual *additive* approach has better performance. In the additive approach, output contexts describe what was *used* instead of what is *left*. To synthesise a term with multiple subterms (e.g. pairs), the output contexts of each premise are added using the semiring addition applied pointwise on contexts to produce the conclusion output. For pairs this looks like:

$$\frac{\Gamma \vdash A \Rightarrow^{+} t_1 \mid \Delta_1 \qquad \Gamma \vdash B \Rightarrow^{+} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} (t_1, t_2) \mid \Delta_1 + \Delta_2}\ \otimes^{+}_{\text{INTRO}}$$

The whole of Γ is used to synthesise both premises. For example, for goal $A \otimes A$:

$$\dfrac{\begin{array}{c} x :_r A, y :_s B \vdash A \Rightarrow^+ x \mid x :_1 A, y :_0 B \\ x :_r A, y :_s B \vdash A \Rightarrow^+ x \mid x :_1 A, y :_0 B \end{array}}{x :_r A, y :_s B \vdash A \otimes A \Rightarrow^+ (x, x) \mid x :_{1+1} A, y :_0 B} \otimes^+_{\textsc{Intro}} \qquad \text{(example)}$$

Synthesis rules for binders check whether the output context describes use that is within the grades given by Γ, i.e., that synthesised terms are *well-resourced.*

Both subtractive and additive approaches avoid having to split the incoming context Γ prior to synthesising subterms. In LGM, we evaluated both resource management strategies in a synthesis tool for a subset of Granule's 'linear base' system, finding that in most cases, the additive strategy was more efficient for use in program synthesis with grades as it involves solving less complex predicates; the subtractive approach typically incurs higher overhead due to the existentially-derived notion of subtraction seen above. We therefore take the additive approach to resource management.

LGM developed our approach for the linear λ-calculus with products, coproducts, and semiring-graded modalities. Here, we instead consider a graded calculus without a linear base but where all assumptions are graded and function types therefore incorporate a grade. Furthermore, our approach permits synthesis for user-defined recursive ADTs to address more real-world problems.

3 Core Calculus

We define a core language with graded types, drawing from the coeffect calculus of Petricek et al. [49], Quantitative Type Theory (QTT) [39, 5] and other graded dependent type theories [42] (omitting dependent types from our language), the calculus of Abel and Bernardy [1], and the core of the linear types extension to Haskell [8]. This calculus shares much in common with languages based on linear types, such as the graded monadic-comonadic calculus of [18], generalisations of Bounded Linear Logic [12, 19], and Granule [45] in its original 'linear base' form.

Our target calculus extends the λ-calculus with grades and a graded necessity modality as well as recursive algebraic data types. Parameterising the calculus is a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ where pre-ordering requires that $+$ and $*$ are monotonic wrt. $\sqsubseteq$. Throughout r, s range over $\mathcal{R}$. The syntax of types is:

$$\begin{array}{rll} A, B ::= & A^r \to B \mid \Box_r A \mid K\,\overline{A} \mid \mu X.A \mid X \mid \alpha & (\textit{types}) \\ K ::= & \mathsf{Unit} \mid \otimes \mid \oplus & (\textit{type constructors}) \\ \tau ::= & \forall \overline{\alpha : \kappa}.A & (\textit{type schemes}) \end{array}$$

The function space $A^r \to B$ annotates the input type with a *grade* $r \in \mathcal{R}$. The graded necessity modality $\Box_r A$ is similarly annotated/indexed with a grade r. Type constructors K include the multiplicative linear products and units, additive coproducts, and is extended by names of user-defined ADTs in the implementation. Constructors are applied to zero or more type parameters written $\overline{A}$. Recursive types $\mu X.A$ are equi-recursive with type recursion variables X. Data constructors and other top-level definitions are typed by type schemes τ (rank-1 polymorphic types), which bind a set of kind-annotated universally quantified type variables $\overline{\alpha : \kappa}$ à la ML [40]. Subsequently, types may contain type variables α. Kinds κ are standard, given in the appendix [28].

The term syntax comprises the λ-calculus, a *promotion* construct $[t]$ which introduces a graded modality, data constructors $(C\, t_1 \,...\, t_n)$, and elimination by **case** expressions with patterns p, where $[p]$ eliminates graded modalities:

$$t ::= x \mid \lambda x.t \mid t_1\, t_2 \mid [t] \mid C\, t_1 \,...\, t_n \mid \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; ...; p_n \mapsto t_n \qquad (\textit{terms})$$
$$p ::= x \mid \_ \mid [p] \mid C\, p_1 \,...\, p_n \qquad (\textit{patterns})$$

Example 1. In the type system (below), the k-combinator is typed as on the left:

$$\begin{array}{ll} k : A^1 \rightarrow B^0 \rightarrow A & k' : (A \times \Box_0 B)^r \rightarrow \Box_r A \\ k = \lambda x.\lambda y.x & k' = \lambda p.\textbf{case}\ p\ \textbf{of}\ (x, y) \mapsto \textbf{case}\ y\ \textbf{of}\ [y'] \mapsto [x] \end{array}$$

On the right, an uncurried version uses graded modalities. The argument pair uses a graded modality to capture that the B part is not used. This graded modal argument is eliminated by the second **case** with pattern $[y']$ binding y' with grade 0, indicating it is unused. The return result is of graded modal type with some grade r which is introduced by promotion $[x]$. Promotion propagates its grade to its dependencies, i.e., the parameter p must also have grade r.

A useful semiring is of *security levels* [18, 1], e.g., $\mathcal{R} = \{\mathsf{Private}, \mathsf{Public}\}$ where $\mathsf{Private} \sqsubseteq \mathsf{Public}$, $+ = \wedge$ with $0 = \mathsf{Private}$, and $* = \vee$ with $1 = \mathsf{Public}$. In the above example, the second argument to k would thus be $\mathsf{Private}$. If the return result of k' is for public consumption, i.e., $r = \mathsf{Public}$, then the argument must also be public, with the private component B not usable in the result.

Figure 1 defines the typing judgments of the form $\Sigma; \Gamma \vdash t : A$ assigning a type A to a term t under type variables Σ. For such judgments we say that t is both *well typed* and *well resourced* to highlight the role of grading in accounting for resource use via the grades. Contexts Γ are given by:

$$\Delta, \Gamma ::= \emptyset \mid \Gamma, x :_r A \qquad (\textit{contexts})$$

That is, a context may be empty $\emptyset$ or extended with a *graded* assumption $x :_r A$. Graded assumptions must be used in a way which adheres to the grade r. Structural exchange is permitted, allowing a context to be arbitrarily reordered. A global context D parameterises the system, containing top-level definitions and data constructors annotated with type schemes. A context of kind annotated type variables Σ is used for kinding and when instantiating a type scheme from D. Appendix A gives the (standard) kinding relation [28].

Variables are typed (rule VAR) in a context where the variable x has grade 1 denoting its single use here. All other variable assumptions are given the grade of the 0 semiring element (providing *weakening*), using *scalar multiplication*:

Definition 1 (Scalar multiplication). *For a context Γ then $r \cdot \Gamma$ scales each assumption by grade r, where $r \cdot \emptyset = \emptyset$ and $r \cdot (\Gamma, x :_s A) = (r \cdot \Gamma), x :_{r \cdot s} A$.*

$$\frac{\Sigma \vdash A : \mathsf{Ty}}{\Sigma; 0 \cdot \Gamma, x :_1 A \vdash x : A}\ \textsc{Var} \qquad \frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \qquad \Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; 0 \cdot \Gamma \vdash x : A}\ \textsc{Def}$$

$$\frac{\Sigma; \Gamma, x :_r A \vdash t : B}{\Sigma; \Gamma \vdash \lambda x.t : A^r \to B}\ \textsc{Abs} \qquad \frac{\Sigma; \Gamma_1 \vdash t_1 : A^r \to B \qquad \Gamma_2 \vdash t_2 : A}{\Sigma; \Gamma_1 + r \cdot \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{App}$$

$$\frac{\Sigma; \Gamma \vdash t : A}{\Sigma; r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{Pr} \qquad \frac{\Sigma; \Gamma, x :_r A, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Sigma; \Gamma, x :_s A, \Gamma' \vdash t : B}\ \textsc{Approx}$$

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.B_1'^{\,q_1} \to \ldots \to B_n'^{\,q_n} \to K\,\overline{A'}) \in D \\ \Sigma \vdash B_1{}^{q_1} \to \ldots \to B_n{}^{q_n} \to K\,\overline{A} = \text{inst}(\forall \overline{\alpha : \kappa}.B_1'^{\,q_1} \to \ldots \to B_n'^{\,q_n} \to K\,\overline{A'})\end{array}}{\Sigma; 0 \cdot \Gamma \vdash C : B_1{}^{q_1} \to \ldots \to B_n{}^{q_n} \to K\,\overline{A}}\ \textsc{Con}$$

$$\frac{\Sigma; \Gamma \vdash t : A \qquad \Sigma; r \vdash p_i : A \rhd \Delta_i \qquad \Sigma; \Gamma', \Delta_i \vdash t_i : B}{\Sigma; r \cdot \Gamma + \Gamma' \vdash \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; \ldots; p_n \mapsto t_n : B}\ \textsc{Case}$$

$$\frac{\Sigma; \Gamma \vdash t : A[\mu X.A/X]}{\Sigma; \Gamma \vdash t : \mu X.A}\ \mu_1 \qquad \frac{\Sigma; \Gamma \vdash t : \mu X.A}{\Sigma; \Gamma \vdash t : A[\mu X.A/X]}\ \mu_2 \qquad \frac{\overline{\alpha : \kappa}; \emptyset \vdash t : A}{\emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A}\ \textsc{TopLevel}$$

Fig. 1: Typing rules

Top-level definitions (Def) must be present in the global definition context D, with the type scheme $\forall \overline{\alpha : \kappa}.A'$. The type A results from instantiating all of the universal variables to types via the judgment $\Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')$ in a standard way as in Algorithm W [40]. Relatedly, the TopLevel rule types top-level definitions with polymorphic type schemes (corresponding to the generalisation rule [40]). Reading bottom up, universally quantified type variables are added to the type variable context to form the type A of the definition term t.

Abstraction (Abs) binds a variable x which is used in the body t according to grade r and thus this grade is captured onto the function arrow in the conclusion. Relatedly, application (App) scales the context Γ_2 of the argument term t_2 by the grade of the function arrow r since t_2 is used according to r in $t_1\, t_2$. To this scaled context is 'added' the context Γ_1 of the function term, via $+$ defined:

Definition 2 (Context addition). *For contexts Γ_1, Γ_2, then $\Gamma_1 + \Gamma_2$ computes the pointwise addition using semiring addition (providing* contraction*), where:*

$$\Gamma + \emptyset = \Gamma \qquad (\Gamma_1, x :_r A) + (\Gamma_2, x :_s A) = (\Gamma_1 + \Gamma_2), x :_{r+s} A$$
$$(\Gamma_1, x :_r A) + \Gamma_2 = (\Gamma_1 + \Gamma_2), x :_r A \qquad \text{if } x \notin \mathsf{dom}(\Gamma_2)$$

For example, $(x :_1 A, y :_0 B) + x :_1 A = x :_{(1+1)} A, y :_0 B$. The operation is commutative and undefined if the type of a variable differs in two contexts.
Introduction of graded modalities is achieved via *promotion* (Pr rule) where grade r is propagated to the assumptions in Γ through the scaling of Γ by r. Approximation (Approx) allows a grade r to be converted to grade s provided that s *approximates* r as defined by the pre-order relation $\sqsubseteq$. This relation is occasionally lifted pointwise to contexts: we write $\Gamma \sqsubseteq \Gamma'$ to mean that Γ' over-approximates Γ, i.e., for all $(x :_r A) \in \Gamma$ then $(x :_{r'} A) \in \Gamma'$ and $r \sqsubseteq r'$.

$$\frac{\Sigma \vdash A : \mathsf{Ty}}{\Sigma; r \vdash x : A \rhd x :_r A}\ \textsc{PVar} \qquad \frac{\Sigma; r \cdot s \vdash p : A \rhd \Gamma}{\Sigma; r \vdash [p] : \Box_s A \rhd \Gamma}\ \textsc{PBox}$$

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\overline{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\overline{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\overline{A'}) \\ \Sigma; q_i \cdot r \vdash p_i : B_i \rhd \Gamma_i \qquad |K\,\overline{A}| > 1 \Rightarrow 1 \sqsubseteq r\end{array}}{\Sigma; r \vdash C\,p_1 \ldots p_n : K\,\overline{A} \rhd \overline{\Gamma_i}}\ \textsc{PCon}$$

Fig. 2: Pattern typing rules

Recursion is typed via the μ_1 rule and its inverse μ_2, in a standard way.

Introduction of data types (Con) via a constructor C of a data type $K\,\overline{A}$ (with zero or more type parameters) incurs an instantiation of its polymorphic type scheme from D. Each argument has a grade q_i. Constructors are closed, thus have only zero-use grades in the context by scaling with 0. Elimination of data types (Case) is via pattern matching. Patterns p are typed by the judgement $r \vdash p : A \rhd \Delta$ (Figure 2) stating that pattern p has type A and produces a context of typed binders Δ. The grade r to the left of the turnstile represents grade information arising from usage in the context generated by this pattern.

Variable patterns (PVar) produce a singleton context with $x :_r A$ of the grade r. Pattern matches on data constructors (PCon rule) may have zero or more sub-patterns ($p_1 ... p_n$), each of which is typed under the grade $q_i \cdot r$ (where q_i is the grade of corresponding argument type for the constructor, as defined in D). Additionally, we have the constraint $|K\,\overline{A}| > 1 \Rightarrow 1 \sqsubseteq r$ which witnesses the fact that if there is more than one data constructor for the data type (written $|K\,\overline{A}| > 1$), then r must approximate 1 because pattern matching on a data constructor incurs some usage since it reveals information about that constructor. [2] By contrast, pattern matching on a type with only one constructor cannot convey any information by itself and so no usage requirement is imposed. Finally, elimination of a graded modality (often called *unboxing*) takes place via the PBox rule, with syntax $[p]$. Like PCon, this rule propagates the grade information of the box pattern's type s to the enclosed sub-pattern p, yielding a context with the grades $r \cdot s$. One may observe that PBox (and by extension Pr) could be considered as special cases of PCon (and Con respectively), if we were to treat promotion as a data constructor with the type $A^r \to \Box_r A$. We however chose to keep modal introduction and elimination distinct from constructors.

Example 2. Discussed early, the natural numbers semiring with discrete ordering $(\mathbb{N}, *, 1, +, 0, \equiv)$ counts exactly how many times variables are used. We denote this semiring as $\mathbb{N}_\equiv$. This semiring is less useful in the presence of control-flow, e.g., for multiple branches in a **case** using variables differently. A semiring of natural number intervals [45] is more helpful here. An interval is a pair of natural numbers $\mathbb{N} \times \mathbb{N}$ written $r...s$ for lower bound $r \in \mathbb{N}$ and upper bound by $s \in \mathbb{N}$. Addition is defined pointwise with zero $0 = 0...0$ and multiplication defined as in

[2] A discussion of this additional constraint on grades is given by Hughes et al. [29]

interval arithmetic with $1 = 1...1$ and ordering $r...s \sqsubseteq r'...s' = r' \leq r \wedge s \leq s'$. This semiring allows us to write a function which performs an elimination on a coproduct (assuming $\mathsf{inl} : A^1 \rightarrow A \oplus B$, and $\mathsf{inr} : B^1 \rightarrow A \oplus B$ in D):

$$\begin{aligned}\oplus_{elim} &: (A^1 \rightarrow C)^{0...1} \rightarrow (B^1 \rightarrow C)^{0...1} \rightarrow (A \oplus B)^1 \rightarrow C \\ \oplus_{elim} &= \lambda f.\lambda g.\lambda x.\textbf{case}\ x\ \textbf{of}\ \mathsf{inl}\ y \mapsto f\ y;\ \mathsf{inr}\ z \mapsto g\ z\end{aligned}$$

Example 3. The ! modality of linear logic can be (almost) recovered via the $\{0,1,\omega\}$ semiring where $0 \sqsubseteq \omega$ and $1 \sqsubseteq \omega$. Addition is $r + s = r$ if $s = 0$, $r + s = s$ if $r = 0$, otherwise ω. Multiplication is $r \cdot 0 = 0 \cdot r = 0$, $r \cdot \omega = \omega \cdot r = \omega$ (where $r \neq 0$), and $r \cdot 1 = 1 \cdot r = r$. This semiring expresses linear and non-linear usage, where 1 indicates linear use, 0 requires the value be discarded, and ω acts as linear logic's ! permitting arbitrary use. This is similar to Haskell's multiplicity annotations, although they have no equivalent of a 0 grade, with only `One` and `Many` grades [8]. Some additional restrictions are required on pattern typing to get exactly the behaviour of ! with respect to products [26], not considered here.

Lastly we note that the calculus enjoys admissibility of substitution [1] which is critical in type preservation proofs, and is needed for soundness of synthesis:

Lemma 1 (Admissibility of substitution). *Let $\Delta \vdash t' : A$, then: If $\Gamma, x :_r A, \Gamma' \vdash t : B$ then $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$*

4 Synthesis Calculus

Having defined the target language, we define our synthesis calculus, which uses the *additive* approach to resource management (see Section 2), with judgments:

$$\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta$$

That is, given an input context Γ, for goal type A we can synthesise the term t with output context Δ describing how variables were used in t. As in typing, top-level definitions and data constructors in scope are provided by a set D parameterising the system. Σ is a context of type variables, which we elide when it is simply passed inductively to the premise(s). The context Δ need not use the variables in Γ with the same grades. Instead, the relationship between synthesis and typing is given by the central soundness result, which we state up-front: that synthesised terms are typed by their goal type under their output context:

Theorem 1 (Soundness). *For all pre-ordered semirings $\mathcal{R}$:*

1. *For all contexts Γ and Δ, types A, terms t:*

$$\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta \quad \implies \quad \Sigma; \Delta \vdash t : A$$

2. *At the top-level, for all type schemes $\forall \overline{\alpha : \kappa}.A$ and terms t then:*

$$\emptyset; \emptyset \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \emptyset \quad \implies \quad \emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A$$

Appendix D of the additional material provides the soundness proof [28], which in part resembles a translation from sequent calculus to natural deduction, but also with the management of grades between synthesis and type checking.

The first part of soundness on its own does not guarantee that a synthesised program t is *well resourced*, i.e., the grades in Δ may not be approximated by grades in Γ. For example, for semiring $\mathbb{N}_{\equiv}$ a valid judgement is:

$$x :_2 A \vdash A \Rightarrow x \mid x :_1 A$$

i.e., for goal A, if x has type A in the context then we synthesis x as the resulting program, regardless of the grades. Such a synthesis judgement may be part of a larger derivation in which the grades eventually match due to a further subderivation, e.g., using x again and thus total usage for x is eventually 2 as prescribed by the input context. However, at the level of an individual judgement we do not guarantee that the synthesised term is well-resourced with respect to the input context. A reasonable *pruning condition* to assess whether any synthesis judgement is *potentially* well-resourced is $\exists \Delta'.(\Delta + \Delta') \sqsubseteq \Gamma$, i.e., there is some additional usage Δ' (that might come from further on in the synthesis process) that 'fills the gap' in resource use to produce $\Delta + \Delta'$ which is overapproximated by Γ. In this example, $\Delta' = x :_1 A$ would satisfy this constraint, explaining that there is some further possible single usage which will satisfy the incoming grade. However, our previous work on graded linear types showed that excessive pruning at every step becomes too costly in a general setting [27]. Instead, we apply such pruning more judiciously, only requiring that variable use is well-resourced at the point of synthesising binders. Therefore synthesised closed terms *are* always well-resourced (second part of the soundness theorem).

We next present the synthesis calculus in stages. Each type former of the core calculus (with the exception of type variables) has two corresponding synthesis rules: a right rule for introduction (labelled R) and a left rule for elimination (labelled L). We frequently apply the algorithmic reading of the judgments, where meta variables to the left of $\Rightarrow$ are inputs (i.e., context Γ and goal type A) and terms to the right of $\Rightarrow$ are outputs (i.e., the synthesised term t and the usage context Δ). Whilst we largely present the approach here in abstract terms, via the synthesis judgments, we highlight some choices made in our implementation (e.g., heuristics applied in the algorithmic version of the rules).

4.1 Core Synthesis Rules

Top-level We begin with synthesis from a type scheme goal (which is technically a separate judgment form), providing the entry-point to synthesis:

$$\frac{\overline{\alpha : \kappa}; \emptyset \vdash A \Rightarrow t \mid \emptyset}{\emptyset; \emptyset \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \emptyset}\ \textsc{TopLevel}$$

The universally-quantified type variables $\overline{\alpha : \kappa}$ are thus added to the type variable context of the premise (note, type variables are only equal to themselves).

Variables For any goal type A, if there is a variable in the context matching this type then it can be synthesised for the goal, given by a terminal rule:

$$\frac{\Sigma \vdash A : \mathsf{Ty}}{\Sigma; \Gamma, x :_r A \vdash A \Rightarrow x \mid 0 \cdot \Gamma, x :_1 A}\ \textsc{Var}$$

Said another way, to synthesise the use of a variable x, we require that x be present in the input context Γ. The output context here then explains that only variable x is used: it consists of the entirety of the input context Γ scaled by grade 0 (using definition 1), extended with $x :_1 A$, i.e. a single usage of x as denoted by the 1 element of the semiring. Maintaining this zeroed Γ in the output context simplifies subsequent rules by avoiding excessive context membership checks.

The Var rule permits synthesis of terms which may not be well-resourced, e.g., if $r = 0$, the rule still synthesises a use of x. As discussed at this section's start, this may be locally ill-resourced, but is acceptable at the global level as we check that an assumption has been used correctly when it is bound. This reduces the number of intermediate theorems that need solving (previously shown to be expensive [27], especially since the variable rule is applied very frequently), but increases the number of paths that are ill-resourced so must be pruned later.

The use of a top-level polymorphic function is synthesised if it can be instantiated to match the goal type:

$$\frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \qquad \Sigma \vdash A = \mathrm{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; \Gamma \vdash A \Rightarrow x \mid 0 \cdot \Gamma}\ \textsc{Def}$$

For example, assuming *flip* : $\forall c : \mathrm{Type}, d : \mathrm{Type}.(c \otimes d)^1 \to (d \otimes c) \in D$ then *flip* is synthesised for a goal type of $(K_1 \otimes K_2)^1 \to (K_2 \otimes K_1)$ for some type constants K_1 and K_2, via the instantiation $\emptyset \vdash (K_1 \otimes K_2)^1 \to (K_2 \otimes K_1) = \mathrm{inst}(\forall c : \mathrm{Type}, d : \mathrm{Type}.(c \otimes d)^1 \to (d \otimes c))$.

Recursion is provided by populating D with the name and type of the definition currently being synthesised for (see Section 4.2 for implementation details).

Functions Synthesis from function types is handled by the $\to_\mathrm{R}$rule:

$$\frac{\Gamma, x :_q A \vdash B \Rightarrow t \mid \Delta, x :_r A \qquad r \sqsubseteq q}{\Gamma \vdash A^q \to B \Rightarrow \lambda x.t \mid \Delta}\ \to_\mathrm{R}$$

Reading bottom up, to synthesise a term of type $A^q \to B$ in context Γ we first extend the context with a fresh variable assumption $x :_q A$ and synthesise a term of type B that will ultimately become the body of the function. The type $A^q \to B$ conveys that A must be used according to q in our term for B. The fresh variable x is passed to the premise of the rule using the grade of the binder: q. The x must then be used to synthesise a term t with q usage. In the premise, after synthesising t we obtain an output context $\Delta, x :_r A$. As mentioned, the Var rule ensures that x is present in this context, even if it was not used in the synthesis of t (e.g., $r = 0$). The rule ensures the usage of bound term (r) in t does not violate the input grade q via the requirement that $r \sqsubseteq q$ i.e. that r is *approximated by* q. If met, Δ becomes the output context of the rule's conclusion.

Function application is synthesised from functions in the context (a left rule):

$$\frac{\begin{array}{l}\Gamma, x :_{r_1} A^q \rightarrow B, y :_{r_1} B \vdash C \Rightarrow t_1 \mid \Delta_1, x :_{s_1} A^q \rightarrow B, y :_{s_2} B \\ \Gamma, x :_{r_1} A^q \rightarrow B \vdash A \Rightarrow t_2 \mid \Delta_2, x :_{s_3} A^q \rightarrow B\end{array}}{\Gamma, x :_{r_1} A^q \rightarrow B \vdash C \Rightarrow [(x\, t_2)/y]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x :_{s_2 + s_1 + (s_2 \cdot q \cdot s_3)} A^q \rightarrow B} \rightarrow_{\mathrm{L}}$$

Reading bottom up, the input context contains an assumption of function type $x :_{r_1} A^q \rightarrow B$. An application of x can be synthesised if an argument t_2 can be synthesised for the input type A (second premise). The goal type C is synthesised (first premise), under the assumption of a result of type B bound to y. In the conclusion, a term is synthesised which substitutes in t_1 the result placeholder variable y for the application $x\, t_2$.

We explain the concluding output context in two stages. Firstly, the output context Δ_1 of the first premise is added to a scaled Δ_2. Since Δ_2 are the resources used by the synthesised argument t_2, this context is scaled by q as t_2 is used according to q by x as per its type. This context is further scaled by s_2 which is the usage of the entire application $x\, t_2$ inside t_1 as given by the output grade for y in the first premise. Secondly, the output context calculates the use of x used in the application itself and potentially also by both premises (which differs from LGM's treatment of synthesis in a linear setting). Apart from application, x may be used also to synthesise the argument t_2, calculated as grade s_3 in the second premise. Thus, the application accrues $q \cdot s_3$ use. Furthermore as the result y is used according to s_2, we must further scale by s_2, obtaining $s_2 \cdot q \cdot s_3$. To this we must also add the additional usage of x in the first premise s_1 as well as the use of x in actually performing application, which is 1 scaled by s_2 to account for the usage of its result, thus obtaining the output grade for x. Following the soundness proof for this rule (Appendix D) can be instructive.

The declarative rule above does not imply an ordering of whether t_1 or t_2 is synthesised first. As a heuristic, the implementation first attempts to synthesise t_1 assuming $y :_{r_1} B$ according to the first premise to avoid possibly unnecessary work if no term can be synthesised anyway for C.

Example 4. Let $T = (A \otimes A)^{0..1} \rightarrow A$ type an assumption fst in a use of $\rightarrow_{\mathrm{L}}$:

$$\frac{\begin{array}{l} z :_s A, \mathsf{fst} :_r T, y :_r A \vdash A \otimes A \Rightarrow (y, y) \mid z :_0 A, \mathsf{fst} :_0 T, y :_2 A \\ z :_s A, \mathsf{fst} :_r T \qquad\quad \vdash A \Rightarrow (z, z) \mid z :_2 A, \mathsf{fst} :_0 T \end{array}}{z :_s A, \mathsf{fst} :_r T \vdash A \otimes A \Rightarrow (\mathsf{fst}\,(z, z), \mathsf{fst}\,(z, z)) \mid z :_{0+2 \cdot (0..1) \cdot 2} A, \mathsf{fst} :_{2+0+(2 \cdot (0..1) \cdot 0)} T}$$

In this instantiation of the $(\rightarrow_{\mathrm{L}})$ rule, $q = 0..1$ and $s_1 = s_3 = 0$, i.e., the function fst is not used in the subterms, and $s_2 = 2$, i.e., the result y of fst is used twice. In the conclusion then, z then has output grade $0 + 2 \cdot (0..1) \cdot 2 = 0..4$, i.e., it is used up to four times and fst has grade $2..2$, i.e., it is used twice.

Graded Modalities Graded modalities are introduced through the $\Box_{\mathrm{R}}$rule, synthesising a promotion $[t]$ for some graded modal type $\Box_r A$:

$$\frac{\Gamma \vdash A \Rightarrow t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow [t] \mid r \cdot \Delta} \Box_{\mathrm{R}}$$

The premise synthesises term t from A with output context Δ. In the conclusion, Δ is scaled by the grade r of the goal type since $[t]$ must use t as r requires.

Grade elimination (*unboxing*) takes place via pattern matching in **case**:

$$\frac{\begin{array}{l}\Gamma, y :_{r \cdot q} A, x :_r \Box_q A \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \\ \exists s_3.\, s_1 \sqsubseteq s_3 \cdot q \sqsubseteq r \cdot q\end{array}}{\Gamma, x :_r \Box_q A \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ [y] \rightarrow t \mid \Delta, x :_{s_3 + s_2} \Box_q A}\ \Box_{\mathrm{L}}$$

To eliminate an assumption x of graded modal type $\Box_q A$, we bind a fresh assumption the premise: $y :_{r \cdot q} A$. This assumption is graded with $r \cdot q$: the grade from the assumption's type multiplied by the grade of the assumption itself. As with previous elimination rules, x is rebound in the rule's premise. A term t is then synthesised resulting in the output context $\Delta, y :_{s_1} A, x :_{s_2} \Box_q A$, where s_1 and s_2 describe how y and x were used in t. The second premise ensures that the usage of y is well-resourced. The grade s_3 represents how much the usage of y inside t contributes to the overall usage of x. The constraint $s_1 \sqsubseteq s_3 \cdot q$ conveys the fact that q uses of y constitutes a single use of x, with the constraint $s_3 \cdot q \sqsubseteq r \cdot q$ ensuring that the overall usage does not exceed the binding grade. For the output context of the conclusion, we simply remove the bound y from Δ and add x, with the grade $s_2 + s_3$ representing the total usage of x in t.

Data Types The synthesis of introduction forms for data types is by the $\mathrm{C_R}$rule:

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \rightarrow \ldots \rightarrow {B'_n}^{q_n} \rightarrow K\,\overline{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \rightarrow \ldots \rightarrow {B_n}^{q_n} \rightarrow K\,\overline{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \rightarrow \ldots \rightarrow {B'_n}^{q_n} \rightarrow K\,\overline{A'}) \\ \Sigma; \Gamma \vdash B_i \Rightarrow t_i \mid \Delta_i\end{array}}{\Sigma; \Gamma \vdash K\,\overline{A} \Rightarrow C\, t_1 \ldots t_n \mid 0 \cdot \Gamma + (q_1 \cdot \Delta_1) + \ldots + (q_n \cdot \Delta_n)}\ \mathrm{C_R}$$

where D is the set of data constructors in global scope, e.g., coming from ADT definitions, including here products, unit, and coproducts with $(,) : A^1 \rightarrow B^1 \rightarrow A \otimes B$, $\mathsf{unit} : \mathsf{Unit}$, $\mathsf{inl} : A^1 \rightarrow A \oplus B$, and $\mathsf{inr} : B^1 \rightarrow A \oplus B$.

For a goal type $K\,\overline{A}$ where K is a data type with zero or more type arguments (denoted by the vector $\overline{A}$), then a constructor term $C\, t_1 \ldots t_n$ for $K\,\overline{A}$ is synthesised. The type scheme of the constructor in D is first instantiated (similar to Def rule), yielding a type ${B_1}^{q_1} \rightarrow \ldots \rightarrow {B_n}^{q_n} \rightarrow K\,\overline{A}$. A sub-term is then synthesised for each of the constructor's arguments t_i in the third premise (which is repeated for each instantiated argument type B_i), yielding output contexts Δ_i. The output context for the rule's conclusion is obtained by performing a context addition across all the output contexts generated from the premises, where each context Δ_i is scaled by the corresponding grade q_i from the data constructor in D capturing the fact that each argument t_i is used according to q_i.

Data type elimination synthesises **case** expressions, pattern matching on each data constructor of the goal data type $K\,\overline{A}$, with various constraints on grades. In the rule, we use the least-upper bound (lub) operator $\sqcup$ on grades, which is defined wrt. $\sqsubseteq$ and may not always be defined:

$$\dfrac{\begin{array}{l}(C_i : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \rightarrow ... \rightarrow {B'_n}^{q_n} \rightarrow K\,\overline{A'}) \in D \qquad \Sigma \vdash K\,\overline{A} : \mathsf{Ty} \\ \Sigma \vdash {B_1}^{q_1} \rightarrow ... \rightarrow {B_n}^{q_n} \rightarrow K\,\overline{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \rightarrow ... \rightarrow {B'_n}^{q_n} \rightarrow K\,\overline{A'}) \\ \Sigma; \Gamma, x :_r K\,\overline{A}, y^i_1 :_{r \cdot q^i_1} B_1, ..., y^i_n :_{r \cdot q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\overline{A}, y^i_1 :_{s^i_1} B_1, ..., y^i_n :_{s^i_n} B_n \\ \exists {s'}^i_j .\, s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \qquad s_i = {s'}^i_1 \sqcup ... \sqcup {s'}^i_n \qquad |K\,\overline{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m\end{array}}{\Sigma; \Gamma, x :_r K\,\overline{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y^i_1 ... y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup ... \sqcup \Delta_m), x :_{\bigsqcup r_i + \bigsqcup s_i} K\,\overline{A}}\ \mathrm{C_L}$$

where $1 \leq i \leq m$ indexes data constructors of which there are m (i.e., $m = |K\,\overline{A}|$) and $1 \leq j \leq n$ indexes arguments of the i^{th} data constructor, thus n depends on i. The rule considers data constructors where $n > 0$ for brevity.

The relevant data constructors C_i are retrieved from the global scope D in the first premise. Each polymorphic type scheme is instantiated to a monomorphic type. The monomorphised type for each i is a function from constructor arguments $B_1 \ldots B_n$ to the applied type constructor $K\,\overline{A}$. For each C_i, we synthesise a term t_i from this result type $K\,\overline{A}$, binding the data constructor's argument types as fresh assumptions to be used in the synthesis of t_i. The grades of each argument are scaled by r. This follows the pattern typing rule for constructors; a pattern match under some grade r must bind assumptions that have the capability to be used according to r. The assumption being eliminated $x :_r K\,\overline{A}$ is also included in the premise's context (as in $\rightarrow_{\mathrm{L}}$) as we may perform additional eliminations on the current assumption subsequently.

The output context for each branch can be broken down into three parts:

1. Δ_i contains any assumptions from Γ were used to construct t_i;
2. $x :_{r_i} K\,A$ describes how the assumption x was used;
3. $y^i_1 :_{s^i_1} B_1, ..., y^i_n :_{s^i_n} B_n$ describes how each assumption y^i_j bound in the pattern match was used in t_i according to grade s^i_j.

For the concluding output context, we take the least-upper bound of the shared output contexts Δ_i of the branches. This is extended with the grade for x which requires some calculation. For each bound assumption, we generate a fresh grade variable ${s'}^i_j$ which represents how that variable was used in t_i after factoring out the multiplication by q^i_j. This is done via the constraint in the third premise that $\exists {s'}^i_j .\, s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j$. The lub of ${s'}^i_j$ for all j is then taken to form a grade variable s_i which represents the total usage of x for branch i arising from the use of assumptions bound via the pattern match (i.e., not usage that arises from reusing x explicitly inside t_i). The final grade for x is then the lub of each r_i (the usages of x directly in each branch) plus the lub of each s_i (the usages of the assumptions that were bound from matching on a constructor of x).

*Example 5 (**case** synthesis).* Consider two possible synthesis results:

$$x :_r A \oplus \mathsf{Unit}, y :_s A, z :_{r \cdot q_1} A \vdash A \Rightarrow z \mid x :_0 A \oplus \mathsf{Unit}, y :_0 A, z :_1 A \qquad (1)$$

$$x :_r A \oplus \mathsf{Unit}, y :_s A \vdash A \Rightarrow y \mid x :_0 A \oplus \mathsf{Unit}, y :_1 A \qquad (2)$$

We will plug these into the rule for generating **case** as follows, where Σ has been elided and instead of using the above concrete grades we have used the abstract form of the rule (the two will be linked by equations after):

$$\frac{\begin{array}{ll} \mathsf{some} : (\forall \alpha, \beta : \mathsf{Ty}.\alpha^1 \rightarrow \alpha \oplus \beta) \in D & \Sigma \vdash A^1 \rightarrow A \oplus \mathsf{Unit} = \mathrm{inst}(\forall \alpha, \beta : \mathsf{Ty}.\alpha^1 \rightarrow \alpha \oplus \beta) \\ \mathsf{none} : (\forall \alpha, \beta : \mathsf{Ty}.\alpha \oplus \beta) \in D & \Sigma \vdash A \oplus \mathsf{Unit} = \mathrm{inst}(\forall \alpha, \beta : \mathsf{Ty}.\alpha \oplus \beta) \\ (1)\ \Sigma; x :_r A \oplus \mathsf{Unit}, y :_s A, z :_{r \cdot q_1} A & \vdash A \Rightarrow z \mid x :_0 A \oplus \mathsf{Unit}, y :_0 A, z :_{s_1} A \\ (2)\ \Sigma; x :_r A \oplus \mathsf{Unit}, y :_s A & \vdash A \Rightarrow y \mid x :_0 A \oplus \mathsf{Unit}, y :_1 A \\ \multicolumn{2}{c}{\exists s'_1 .\, s_1 \sqsubseteq s'_1 \cdot q_1 \sqsubseteq r \cdot q_1 \qquad s' = s'_1 \qquad |A \oplus \mathsf{Unit}| \implies 1 \sqsubseteq s_1} \end{array}}{x :_r A \oplus \mathsf{Unit}, y :_s A \vdash A \Rightarrow (\textbf{case}\ x\ \textbf{of}\ \mathsf{some}\, z \rightarrow z; \mathsf{none} \rightarrow y) \mid x :_{(0 \sqcup 0) + s'} A \oplus \mathsf{Unit}, y :_{0 \sqcup 1} A}$$

To unify (1) and (2) with the $\mathrm{C_L}$ rule format $s_1 = 1$ and $q_1 = 1$ (from the type of inl). Applying these equalities to the existential constraint we have

$$\exists s'_1 .\, 1 \sqsubseteq (s'_1 \cdot 1) \sqsubseteq (r \cdot 1) \quad \implies \quad \exists s'_1 .\, 1 \sqsubseteq s'_1 \sqsubseteq r$$

With the natural-number intervals semiring this is satisfied by $s'_1 = 1..1 = s'$ and thus in the output context x has grade 1..1 and y has grade 0..1.

Recursive Types Though μ types are equi-recursive, we define explicit synthesis rules to facilitate the implementation (Section 4.2) where depth information needs to be tracked when employing the following μ_L and μ_R rules:

$$\frac{\Gamma \vdash A[\mu X.A/X] \Rightarrow t \mid \Delta}{\Gamma \vdash \mu X.A \Rightarrow t \mid \Delta}\ \mu_\mathrm{R} \qquad \frac{\Gamma, x :_r A[\mu X.A/X] \vdash B \Rightarrow t \mid \Delta}{\Gamma, x :_r \mu X.A \vdash B \Rightarrow t \mid \Delta}\ \mu_\mathrm{L}$$

To synthesise a recursive data structure of type $\mu X.A$, we must be able to synthesise A with $\mu X.A$ substituted for the recursion variable X in A. For example, if we wish to synthesise a list typed `List a` (where `Cons : a` $\rightarrow$ `List a` $\rightarrow$ `List a`) then when synthesising a `Cons` constructor in the μ_R rule, we must re-apply the μ_R rule to synthesise the recursive argument. Elimination of a value $\mu X.A$ in the context is via the μ_L, which expands the recursive type in the synthesis context.

4.2 Algorithmic Implementation

The calculus presented above serves as a starting point for our implemented synthesis algorithm in Granule. However, the rules are highly non-deterministic with regards their order in which they may be applied. For example, after applying a ($\rightarrow_\mathrm{R}$)-rule, we may choose to apply any of the elimination rules before applying an introduction rule for the goal type. This leads to us exploring a large number of redundant search branches which can be avoided through the application of a technique known as *focusing* [4]. Focusing is a tool from linear logic proof theory based on the idea that some rules are invertible, i.e., whenever the conclusion of the rule is derivable, then so are the premises. In other words, the order in which we apply invertible rules doesn't matter. By fixing a particular ordering on the application of invertible rules, we eliminate much of the non-determinism that arises from trying branches which differ only in the order in which invertible rules are applied. The full focusing versions of the rules from our calculus, and their proof of soundness, can be found in Appendix E [28]. This forms the basis of our implementation with the high-level algorithm given in appendix Figure 5 as a (non-deterministic) finite state machine, which shows the ordering given to the rules under the focussing approach, starting with trying to synthesise function types before switching to eliminations rules, and so on. In standard terminology, our algorithm is ‘top-down’ (see, e.g., [17, 47, 23, 53]), or *goal-directed*, in which we start with a type goal and an input context and progress by gradually building the syntax tree from the empty term following the focussing-ordered rules of our calculus. This contrasts with ‘bottom-up’ approaches [2, 41, 44] which maintain complete programs which can be executed (tested) and combined.

Where transitions are non-deterministic in the algorithm, multiple branches are then explored in synthesis. Our implementation relies on the use of backtracking proof search, leveraging a monadic interface that provides both choice (e.g., between multiple possible synthesis options based on the goal type) and failure (e.g., when a constraint fails to hold) [33]. For every rule that generates a constraint on grades, due to binding ($\Box_\mathrm{L}$, $\rightarrow_\mathrm{R}$, C_L), we compile the constraints to the SMT-lib format [7] which are then discharged by the Z3 SMT solver [43]. If the constraint is invalid then we trigger the failure of this synthesis pathway, triggering backtracking via the “logic” monad [33]. A synthesised program can also be rejected by user (or due to a failing an example, see below) and synthesis then produces an alternate result (what we call a *retry*) via backtracking.

Recursive data structures present a challenge in the implementation. For example, for the list data type, how do we prevent synthesis from applying the μ_L rule, followed by the C_L rule on the `Cons` constructor ad infinitum? We resolve this issue using an *iterative deepening* approach similar to that used by Myth [48]. Programs are synthesised with elimination (and introduction) forms of constructors restricted up to a given depth. If no program is synthesised within these bounds, then the depth limits are incremented. The current depth and the depth limit are part of the state of the synthesiser. Combined with focusing this provides the basis an efficient implementation of the synthesis calculus. Furthermore, to ensure that a synthesised programs terminates, we only permit synthesis of recursive function calls which are *structurally recursive*, i.e., those which apply the recursive definition to a subterm of the function’s inputs [48].

Lastly, after synthesis, a post-synthesis refactoring step runs to simplify terms and produce a more idiomatic style. For example for the k combinator type signature `k : ∀ {a b : Type} . a %1 → b %0 → a` we synthesis the term: `k = λx → λy → x`. Our refactoring procedure collects the outermost abstractions of a synthesised term and transforms them into equation-level patterns with the innermost abstraction body forming the equation body: `k x y = x`. Repeated **case** expressions are also refactored into nested pattern matches, which are part of Granule. For example, nested matching on pairs is simplified to a single **case** with nested pattern matching: `case x of (y1, y2) → case y1 of (z1, z2) → e` is refactored to `case x of ((z1, z2), y2) → e`.

Input-output Examples Further to the implementation described above, we also allow user-defined input-output examples which are checked as part of synthesis.

Our approach is deliberately naïve: we evaluate a fully synthesised candidate program against the inputs and check that the results match the corresponding outputs. Unlike sophisticated example-driven synthesis tools, the examples only influence the search procedure by backtracking on a complete program that doesn't satisfy the examples. This lets us consider the effectiveness of search based primarily around the use of grades (see Section 5). Integrating examples more tightly with the type-and-grade directed approach is further work.

Our implementation augments Granule with first-class syntax for specifying input-output examples, both as a feature for aiding synthesis but also for aiding documentation that is type checked (and therefore more likely to stay consistent with a code base as it evolves). Synthesis specifications are written in Granule directly above a program hole (written using `?`) using the `spec` keyword. The input-output examples are then listed per-line. For example, one of benchmark programs (Section 5) for the length of a list is specified as:

```
1 length : ∀ a . List a %0..∞ → N
2 spec length (Cons 1 (Cons 1 Nil)) = S (S Z);
3      length
4 length = ?
```

Any synthesised definition must then behave according to this example.

In a `spec` block, a user can also specify the names of functions in scope which are to be taken as the available definitions (set D in the formal specification). For example, line 4 above specifies that `length` can be used here (i.e., recursively).

5 Evaluation

In evaluating our approach and tool, we made the following hypotheses:

H1. (**Expressivity; less consultation**) The use of grades in synthesis results in a synthesised program that is more likely to have the behaviour desired by the user; the user needs to request fewer alternate synthesised results (*retries*) and thus is consulted less in order to arrive at the desired program.

H2. (**Expressivity; fewer examples**) Grade-and-type directed synthesis requires fewer input-output examples to arrive at the desired program compare with a purely type-driven approach.

H3. (**Performance; more pruning**) The ability to prune resource-violating candidate programs from the search tree leads to a synthesised program being found more quickly when synthesised from a graded type compared with the same type but without grades (purely type-driven approach).

5.1 Methodology

To evaluate our approach, we collected a suite of benchmarks comprising graded type signatures for common transformations on structures such as lists, streams, booleans, option ('maybe') types, unary natural numbers, and binary trees. A representative sample of benchmarks from the Myth synthesis tool [47] are included alongside a variety of other programs one might write in a graded setting. Benchmarks are categorised based on the main data type, with an additional miscellaneous category. Appendix C lists type schemes for all benchmarks [28]. To compare, in various ways, our grade-and-type-directed synthesis to traditional type-directed synthesis, each benchmark signature is also "de-graded" by replacing all grades in the type with `Any` which is the only element of the singleton `Cartesian` semiring in Granule. When synthesising in this semiring, we can forgo discharging grade constraints in the SMT solver entirely. Thus, synthesis for Cartesian grades degenerates to type-directed synthesis following our rules.

To assess hypothesis 1 (grade-and-type directed leads to less consultation / more likely to synthesise the intended program) we perform grade-and-type directed synthesis on each benchmark problem and type-directed synthesis on the corresponding de-graded version. For the de-graded versions, we record the number of retries N needed to arrive at a well-resourced answer by type checking the output programs against the original graded type signature, retrying if the program is not well-typed (essentially, not well-resourced). This checks whether a program is 'as intended' without requiring input from a user. In each case, we also compared whether the resulting programs from synthesis via graded-and-type directed vs. type-directed with retries (on non-we were equivalent.

To assess hypothesis 2 (graded-and-type directed requires fewer examples than type-directed), we run the de-graded (Cartesian) synthesis with the smallest set of examples which leads to the model program being synthesised (without any retries). To compare across approaches to the state-of-the-art type-directed approach, we also run a separate set of experiments comparing the minimal number of examples required to synthesise in Granule (with grades) vs. Myth.

To assess hypothesis 3 (grade-and-type-directed faster than type-directed) we compare performance in the graded setting to the de-graded Cartesian setting. Comparing our tool for speed against another type-directed (but not graded-directed) synthesis tool such as Myth is likely to be largely uninformative due to differences in implementation (engineering artefacts) obscuring meaningful comparison. Thus, we instead compare timings for the graded and de-graded approach within Granule. This normalises implementation artefacts as the two approaches vary only in the use of SMT solving to prune ill-resourced programs (in the graded approach). We also record the number of search paths taken (over all retries) to assess the level of pruning in the graded vs de-graded case.

We ran our synthesis tool on each benchmark for both the graded type and the de-graded Cartesian case, computing the mean after 10 trials for timing data. Benchmarking was carried out using version 4.12.1 of Z3 [43] on an M1 MacBook Air with 16 GB of RAM. A timeout limit of 10 seconds was set for synthesis.

5.2 Results and Analysis

Table 1 records the results comparing grade-and-type synthesis vs. the Cartesian (de-graded) type-directed synthesis. The left column gives the benchmark name, number of top-level definitions in scope that can be used as components (size of the synthesis context) labelled CTXT, and the minimum number of examples needed (#/Exs) to synthesise the Graded and Cartesian programs. In the Cartesian setting, where grade information is not available, if we forgo type-checking a candidate program against the original graded type then additional input-output examples are required to provide a strong enough specification such that the correct program is synthesised (see H3). The number of additional examples is given in parentheses for those benchmarks which required these additional examples to synthesise a program in the Cartesian setting.

Each subsequent results column records: whether a program was synthesised successfully ✓ or not × (due to timeout or no solution found), the mean synthesis time (μT) or if timeout occurred, and the number branching paths (Paths) explored in the synthesis search space.

The first results column (Graded) contains the results for graded synthesis. The second results column (Cartesian + Graded type-check) contains the results for synthesising in the Cartesian (de-graded) setting, using the same examples set as the Graded column, and recording the number of retries (consultations of the type-checker at the end) N needed to reach a well-resourced program. In all cases, the resulting program in the Cartesian case was equivalent to that generated by the graded synthesis, none of which needed any retries (i.e., implicitly $N = 0$ for graded synthesis, i.e., no retries are needed). H1 is confirmed by the fact that N is greater than 0 in 29 out of 46 benchmarks (60%), i.e., the Cartesian case does not synthesis the correct program first time and needs multiple retries to reach a well-resource program, with a mean of 19.60 retries and a median of 4 retries.

For each row, we highlight the column which synthesised a result the fastest in blue. In 17 of the 46 benchmarks (37%) the graded approach out-performed non-graded synthesis. This contradicts hypothesis 3 somewhat: whilst type-directed synthesis often requires multiple retries (versus no retries for graded) it still out-performs graded synthesis. This is due to the cost of SMT solving which must compile a first-order theorem on grades into the SMT-lib file format, start Z3, and then run the solver. Considerable amounts of system overhead are incurred in this procedure. A more efficient implementation calling Z3 directly (via a dynamic library call) may give more favourable results here. However, H3 is still somewhat supported: the cases in which the graded does outperform the Cartesian are those which involve considerable complexity in their use of grades, such as `stutter`, `inc`, and `bind` for lists, and `sum` for both lists and trees. In each case, the Cartesian column is significantly slower, even timing out for `stutter`; this shows the power of the graded approach. Furthermore, we highlight the column with the smallest number of synthesis paths explored in yellow, observing that the number of paths in the graded case is always the same or less than that those in the Cartesian+graded type check case (apart from Tree stutter). The paths explored are the sometimes the same between Graded and Cartesian synthesis because we use backtracking search even in the Cartesian case where, if an output program fails to type check against the graded type, the search backtracks rather than starting from the beginning. This leads to an equal number of paths in the graded case when solving occurred only at a top-level abstraction. However, paths explored are fewer in the graded case when solving occurs at other binders, e.g., in **case** and unboxing.

Confirming H2, the de-graded setting without graded type checking requires more examples to synthesise the same program as the graded in 20 out of 46 (43%) cases. In these cases, an average of 1.25 additional examples are required. To further interrogate H2, we compare the number of examples required by

				Graded			Cartesian + Graded type-check			
	Problem	Ctxt	#/Exs.		μT (ms)	Paths		μT (ms)	N	Paths
List	append	0	0 (+1)	✓	115.35 (5.13)	130	✓	105.24 (0.36)	8	130
	concat	1	0 (+3)	✓	1104.76 (1.60)	1354	✓	615.29 (1.43)	12	1354
	empty	0	0	✓	5.31 (0.02)	17	✓	1.20 (0.01)	0	17
	snoc	1	1	✓	2137.28 (2.14)	2204	✓	1094.03 (4.75)	8	2278
	drop	1	1	✓	1185.03 (2.53)	1634	✓	445.95 (1.71)	8	1907
	flatten	2	1	✓	1369.90 (2.60)	482	✓	527.64 (1.04)	8	482
	bind	2	0 (+2)	✓	62.20 (0.21)	129	✓	622.84 (0.95)	18	427
	return	0	0 (+1)	✓	19.71 (0.18)	49	✓	22.00 (0.08)	4	49
	inc	1	1	✓	708.23 (0.69)	879	✓	2835.53 (7.69)	24	1664
	head	0	1	✓	68.23 (0.53)	34	✓	20.78 (0.10)	4	34
	tail	0	1	✓	84.23 (0.20)	33	✓	38.59 (0.06)	8	33
	last	1	1 (+1)	✓	1298.52 (1.17)	593	✓	410.60 (6.25)	4	684
	length	1	1	✓	464.12 (0.90)	251	✓	127.91 (0.58)	4	251
	map	1	0 (+1)	✓	550.10 (0.61)	3075	✓	249.42 (0.73)	4	3075
	replicate5	0	0 (+1)	✓	372.23 (0.70)	1295	✓	435.78 (1.06)	4	1295
	replicate10	0	0 (+1)	✓	2241.87 (4.74)	10773	✓	2898.93 (1.47)	4	10773
	replicateN	1	1	✓	593.86 (1.68)	772	✓	108.98 (0.65)	4	772
	stutter	1	0	✓	1325.36 (1.77)	1792	×	Timeout	-	-
	sum	2	1 (+1)	✓	84.09 (0.25)	208	✓	3236.74 (0.87)	192	3623
Stream	build	0	0 (+1)	✓	61.27 (0.45)	75	✓	84.44 (0.49)	4	75
	map	1	0 (+1)	✓	351.93 (0.91)	1363	✓	153.01 (0.37)	0	1363
	take1	0	0 (+1)	✓	34.02 (0.23)	22	✓	19.32 (0.05)	0	22
	take2	0	0 (+1)	✓	110.18 (0.31)	204	✓	89.10 (0.18)	0	208
	take3	0	0 (+1)	✓	915.39 (1.42)	1139	✓	631.47 (1.14)	0	1172
Bool	neg	0	2	✓	209.09 (0.31)	42	✓	168.37 (0.56)	0	42
	and	0	4	✓	3129.30 (2.82)	786	✓	7069.14 (15.91)	0	2153
	impl	0	4	✓	1735.09 (4.31)	484	✓	3000.48 (4.65)	0	1214
	or	0	4	✓	1213.86 (1.02)	374	✓	2867.74 (3.52)	0	1203
	xor	0	4	✓	2865.79 (4.33)	736	✓	7251.38 (32.06)	0	2229
Maybe	bind	0	0 (+1)	✓	159.87 (0.52)	237	✓	55.33 (0.33)	0	237
	fromMaybe	0	0 (+2)	✓	54.27 (0.35)	18	✓	11.58 (0.10)	0	18
	return	0	0	✓	9.89 (0.02)	17	✓	11.49 (0.04)	4	17
	isJust	0	2	✓	69.33 (0.17)	48	✓	22.07 (0.09)	0	48
	isNothing	0	2	✓	102.42 (0.32)	49	✓	31.89 (0.22)	0	49
	map	0	0 (+1)	✓	54.90 (0.22)	120	✓	22.01 (0.10)	0	120
	mplus	0	1	✓	319.64 (0.47)	318	✓	70.98 (0.05)	0	318
Nat	isEven	1	2	✓	1027.79 (1.28)	466	✓	313.77 (0.92)	8	468
	pred	0	1	✓	46.20 (0.18)	33	✓	48.04 (0.13)	8	33
	succ	0	1	✓	115.16 (0.91)	76	✓	156.02 (0.50)	8	76
	sum	1	1 (+2)	✓	1582.23 (3.60)	751	✓	734.38 (1.41)	12	751
Tree	map	1	0 (+1)	✓	1168.60 (1.21)	4259	✓	525.47 (1.31)	4	4259
	stutter	1	0 (+1)	✓	693.44 (1.21)	832	✓	219.91 (1.02)	4	674
	sum	2	3	✓	1477.83 (1.28)	3230	✓	3532.24 (7.19)	192	3623
Misc	compose	0	0	✓	40.27 (0.08)	38	✓	14.53 (0.09)	2	38
	copy	0	0	✓	5.24 (0.04)	21	✓	6.16 (0.10)	2	21
	push	0	0	✓	26.66 (0.18)	45	✓	14.23 (0.13)	2	45

Table 1: Results. μT in *ms* to 2 d.p. with standard sample error in brackets

		Granule	Myth	Smyth
Problem		#/Exs	#/Exs	#/Exs
List	append	0	6	4
	concat	1	6	3
	snoc	1	8	3
	drop	1	13	5
	inc	1	4	2
	head	1	3	2
	tail	1	3	2
	last	1	6	4
	length	1	3	3
	map	0	8	4

		Granule	Myth	Smyth
Problem		#/Exs	#/Exs	#/Exs
List	stutter	0	3	2
	sum	1	3	2
Bool	neg	2	2	2
	and	4	4	3
	impl	4	4	3
	or	4	4	3
	xor	4	4	4
Nat	isEven	2	4	3
	add	1	3	2
	pred	1	3	2
Tree	map	0	7	4

Table 2: Number of examples needed for synthesis, Granule vs. Myth vs. Smyth

Granule (using grades) against the Myth synthesis tool (based on pruning by examples) [47], and the more advanced assertion-based Smyth [36]. We consider the subset of our benchmarks drawn from Myth. Table 2 shows the minimum number of input-output examples needed to synthesise the correct program in Granule, Myth, and Smyth. For all cases, Granule required the same or fewer examples than Myth to synthesis the desired program, requiring fewer examples in 16 out of 21 cases. The disparity in the number of examples required is quite significant in some cases: with 13 examples required by Myth to synthesise *concat* but only 1 example for Granule. Overall, Smyth needed the same or fewer examples than Myth. Granule needed the same or fewer examples than Smyth in 18 out of 21 cases, but in the other 3 cases (and, impl, or) Smyth required 1 fewer example. Overall, the lower number of examples needed in our approach shows the pruning power of grades in synthesis, confirming H2.

We briefly examine one of the more complex benchmarks which uses almost all of our synthesis rules in one program. The `stutter` case (List class) is specified:

```
stutter : ∀ a . List (a [2]) %1..∞ → List a
spec stutter
```

Its input is a list of elements graded by `2`, i.e., must be used twice. The argument list itself must be used at least once but possibly infinitely, suggesting that some recursion will be necessary. This is further emphasised by the `spec`, which states we can use `stutter` itself inside the function. Without grades, synthesis times out. Graded synthesise produces the following in 1325ms (∼1.3 seconds):

```
stutter Nil = Nil;
stutter (Cons [u] z) = (Cons u) ((Cons u) (stutter z))
```

6 Synthesis of Linear Haskell Programs

As part of a growing trend of resourceful types being added to more mainstream languages, Haskell has introduced support for linear types as of GHC 9, using an underlying graded type system which can be enabled as a language extension [8] (called `LinearTypes`). This system is closely related to the calculus here but limited to one semiring. This however presents an opportunity to leverage our tool to synthesise (linear) Haskell programs. Like Granule, grades in Haskell can be expressed as "multiplicities" on function types: `a %r -> b`. The multiplicity r can be either 1 or ω (or polymorphic), with 1 denoting linear usage (also written as `'One`) and ω (`'Many`) for unrestricted use. Similarly, Granule can model linear types using the 0-1-ω semiring (Example 1) [26]. Synthesising Linear Haskell programs then simply becomes a task of parsing a Haskell type into a Granule equivalent, synthesising a term from it, and compiling the synthesised term back to Haskell (which has similar syntax to Granule anyway).

Our implementation includes a prototype synthesis tool using this approach. A synthesis problem takes the form of a Linear Haskell program with a hole, e.g.

```
{-# LANGUAGE LinearTypes #-}
swap :: (a, b) %One -> (b, a)
swap = _
```

We invoke the synthesis tool with `gr --linear-haskell swap.hs` which produces:

```
swap (z, y) = (y, z)
```

Users may make use of lists, tuples, `Maybe` and `Either` data types from Haskell's prelude, as well as user-defined ADTs. Further integration of the tool, as well as support for additional Haskell features such as GADTs is left as future work.

7 Discussion

Comparison with prior work Previously, LGM targeted the linear λ-calculus with graded modalities [27]. In this paper, we instead considered a fully-graded ('graded base') calculus with no linearity: all assumptions are graded and subsequently there is a graded function arrow (not present in the 'linear base' style). This graded calculus matches practical implementations of graded types seen in Idris 2 and Haskell. Furthermore, a key contribution beyond LGM is the handling of recursion, general user-defined (recursive) ADTs, and polymorphism. Due to the pervasive grading, the majority of the synthesis rules are considerably different to LGM. For example, LGM's synthesis of functions is linear, and thus need not handle the complexity of grading (*cf.* $\rightarrow_{\mathrm{L}}$ on p. 13):

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1; \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^{+} t_2; \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1 t_2)/x_2]t_1; (\Delta_1 + \Delta_2), x_1 : A \multimap B} L \multimap^{+} [27]$$

As above, in the linear setting of LGM, many of the constraints and grades handled in this paper are essentially specialised away as equal to 1, with only linear products and coproducts considered. Since grading is potentially more permissive than linearity, elimination rules in our synthesis calculus must also make available an eliminated variable for re-use in every premise, which was not needed in LGM. Furthermore, the power of this paper's **case** rule means there are simple, non-recursive terms we can synthesise which LGM cannot. In particular, synthesis of programs which perform "deep" pattern matching over a graded data structure are not possible in LGM. For example, LGM's approach cannot synthesise a term for $\Box_{0..1}(\alpha, (\alpha, \beta)) \multimap \beta$ as it cannot propagate information from one **case** to another to inherit the grade 0..1 on the pair's components. However, here we can synthesise (in just a few steps, plus refactoring):

```
deep : ∀ a b : Type . (a, (a, b)) %0..1 → b
deep [(_, (_, y))] = y                          -- y inherits grade 0..1
```

Thus, not only does our approach consider a different mode of grading, as well as extending to arbitrary recursive ADTs and recursive functions, it is also more expressive in the interaction between data types and grades.

LGM introduced *additive* and *subtractive* resource management schemes (summarised and re-contextualised in Section 2). Comparative evaluation of LGM showed that constraints from the subtractive approach are typically larger, more complex, and discharged more frequently than in additive synthesis. We concluded that subtractive only ever outperformed additive on purely linear types. Coupled with the fact that the subtractive approach has limitations in the presence of polymorphic grades, we thus adopted the additive scheme, especially in light of us considering more complex programs. Our evaluation of LGM did not given any evidence justifying use of grades for synthesis compared to just using types. Here, we showed that grading significantly reduces the number of paths explored and examples needed when compared with purely type-directed approaches, including in comparison with MYTH [47].

Other Related Work Beyond MYTH, other recent work has extended type-and-example-directed synthesis approaches. SMYTH constrains the search space further by augmenting types with assertions (called 'sketches') to guide synthesis [36]. This techniques involves employing more evaluation during synthesis to generated intermediate input-output examples to prune the search space. They evaluate on a subset of MYTH benchmarks (somewhat similar to our own method here). Whilst we compared our approach (with graded + types + examples considered at the end) to MYTH (with types + examples integrated) to show that grading reduces the number of examples, comparing with the assertion-based approach in SMYTH is further work. Another recent work, BURST, also leverages the MYTH benchmark, but using a 'bottom-up' technique [41] (in contrast to our top-down approach, Section 4.2). The bottom-up approach synthesises a sequence of complete programs which can be refined and tested under an 'angelic semantics'. Whether a bottom-up grade-directed approach could lead to performance improvements is an open question.

Whilst we considered resourceful programming via graded types, other notions of resourceful typing exist, including 'ownership' (e.g., Rust [31]) and related 'uniqueness' (e.g., Clean [51]). Recently, Fiala et al. synthesised Rust programs from a custom program logic *Synthetic Ownership Logic* that integrates a typed approach to Rust ownership with functional specifications, allowing synthesis to follow a deductive approach [16]. There is some philosophical overlap in the resourceful ideas in their approach and ours. Drawing a closer correspondence between Rust-style ownership and grading, to perhaps leverage our resourceful approach to synthesis, is future work. Notably, Marshall and Orchard show that uniqueness types can be implemented as an extension of a linear type theory with a non-linearity modality and uniqueness modality [38]. Further work could adapt our approach to this setting to provide synthesis for uniqueness types as a precursor to the full ownership and borrowing system of Rust.

The dependently-typed language Idris provides automated proof search as part of its implementation [10]. In Idris 2, the core type theory is based on a graded type theory [5, 39] with 0-1-ω semiring (Example 1) and with proof synthesis extended to utilise these grades [11]. This approach has some relation to ours, but in a limited single-semiring setting and restricted in how grades can be leveraged. Our approach is readily applicable to Idris, which is future work.

Conclusion Our work is grounded in the philosophy of type-driven development where the user thinks about the expected behaviour or constraints of a program first, writing the type as a specification. Synthesis is not necessarily about having complicated programs generated but is often about generating straightforward programs to save effort. This is the gain provided by type-directed synthesis in existing languages such as Agda [9] and Idris [10]. Our technique augments this, such that boilerplate code and simple algorithms can be automatically generated, freeing the developer to focus on other parts of a program.

A next step is to incorporate GADTs (Generalised ADTs), i.e., indexed types, into synthesis. Granule provides support for user-defined GADTs, and the interaction between grades and type indices is a key contributor to its expressive power [45]. For example, consider a function that replicates a value a number of times to create a list, typed `rep : ∀ {t : Type} . Int → t % 0..∞ → List t`. Given a standard indexed type of natural numbers `N (n : Nat)` and sized-indexed vectors `Vec (n : Nat) (t : Type)`, a more precise specification can be given as `∀ {n : Nat, t : Type} . N n → t % n → Vec n t` for which the search space could be more effectively pruned by including type indices in synthesis.

We intend to pursue further improvements to our tool to reduce the overhead of SMT solving, integrate examples into the search algorithm itself in the style of Myth [47] and Leon [34], as well as considering possible semiring-dependent optimisations that may be applicable. Another further work is prove completeness of our synthesis calculus which we believe this holds.

With the rise in Large Language Models showing their power at program synthesis [6, 30] the deductive approach still has value, providing correct-by-construction synthesis from specification rather than predicting programs which may violate fine-grained type constraints, e.g., from grades. Future work, and a general challenge for the deductive synthesis community, is to combine the two approaches with the logical engine of the deductive approach guiding prediction.

Data-Availability An artefact supporting the results of this work is available at `http://zenodo.org/records/10511509`.

Acknowledgements Thank you to Frank Pfenning and the anonymous reviewers for their helpful comments, and to Ben Orchard for his help in preparing this work's artefact. This work was supported by an EPSRC grant EP/T013516/1.

Bibliography

[1] Abel, A., Bernardy, J.: A unified view of modalities in type systems. Proc. ACM Program. Lang. **4**(ICFP), 90:1–90:28 (2020). `https://doi.org/10.1145/3408972`, `https://doi.org/10.1145/3408972`

[2] Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. pp. 934–950 (2013). `https://doi.org/10.1007/978-3-642-39799-8_67`, `https://doi.org/10.1007/978-3-642-39799-8_67`

[3] Allais, G.: Typing with Leftovers-A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. In: 23rd International Conference on Types for Proofs and Programs (TYPES 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)

[4] Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2**(3), 297–347 (06 1992). `https://doi.org/10.1093/logcom/2.3.297`

[5] Atkey, R.: Syntax and Semantics of Quantitative Type Theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65 (2018). `https://doi.org/10.1145/3209108.3209189`, `https://doi.org/10.1145/3209108.3209189`

[6] Austin, J., Odena, A., Nye, M.I., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C.J., Terry, M., Le, Q.V., Sutton, C.: Program synthesis with large language models. CoRR **abs/2108.07732** (2021), `https://arxiv.org/abs/2108.07732`

[7] Barrett, C., Stump, A., Tinelli, C., et al.: The smt-lib standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)

[8] Bernardy, J., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang. **2**(POPL), 5:1–5:29 (2018). `https://doi.org/10.1145/3158093`, `https://doi.org/10.1145/3158093`

[9] Bove, A., Dybjer, P., Norell, U.: A brief overview of agda–a functional language with dependent types. In: Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22. pp. 73–78. Springer (2009)

[10] Brady, E.C.: Idris, a general-purpose dependently typed programming language: Design and implementation. J. Funct. Program. **23**(5), 552–593 (2013). `https://doi.org/10.1017/S095679681300018X`, `https://doi.org/10.1017/S095679681300018X`

[11] Brady, E.C.: Idris 2: Quantitative type theory in practice **194**, 9:1–9:26 (2021). `https://doi.org/10.4230/LIPICS.ECOOP.2021.9`, `https://doi.org/10.4230/LIPIcs.ECOOP.2021.9`

[12] Brunel, A., Gaboardi, M., Mazza, D., Zdancewic, S.: A core quantitative coeffect calculus. In: Shao, Z. (ed.) Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8410, pp. 351–370. Springer (2014). https://doi.org/10.1007/978-3-642-54833-8_19

[13] Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theoretical Computer Science **232**(1), 133 – 163 (2000). https://doi.org/https://doi.org/10.1016/S0304-3975(99)00173-5

[14] Choudhury, P., III, H.E., Eisenberg, R.A., Weirich, S.: A graded dependent type system with a usage-aware semantics. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021). https://doi.org/10.1145/3434331, https://doi.org/10.1145/3434331

[15] Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. SIGPLAN Not. **50**(6), 229–239 (jun 2015). https://doi.org/10.1145/2813885.2737977, https://doi.org/10.1145/2813885.2737977

[16] Fiala, J., Itzhaky, S., Müller, P., Polikarpova, N., Sergey, I.: Leveraging rust types for program synthesis. Proceedings of the ACM on Programming Languages **7**(PLDI), 1414–1437 (2023)

[17] Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. ACM SIGPLAN Notices **51**(1), 802–815 (2016)

[18] Gaboardi, M., Katsumata, S.y., Orchard, D., Breuvart, F., Uustalu, T.: Combining Effects and Coeffects via Grading. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. p. 476–489. ICFP 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2951913.2951939, https://doi.org/10.1145/2951913.2951939

[19] Ghica, D.R., Smith, A.I.: Bounded linear types in a resource semiring. In: Shao, Z. (ed.) Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014. Lecture Notes in Computer Science, vol. 8410, pp. 331–350. Springer (2014). https://doi.org/10.1007/978-3-642-54833-8_18

[20] Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1 – 101 (1987). https://doi.org/https://doi.org/10.1016/0304-3975(87)90045-4

[21] Girard, J.Y., Scedrov, A., Scott, P.J.: Bounded linear logic: a modular approach to polynomial-time computability. Theoretical computer science **97**(1), 1–66 (1992)

[22] Green, C.: Application of theorem proving to problem solving. In: Proceedings of the 1st International Joint Conference on Artificial Intelligence. p. 219–239. IJCAI'69, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1969)

[23] Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices **46**(1), 317–330 (2011)

[24] Harland, J., Pym, D.J.: Resource-distribution via boolean constraints. CoRR **cs.LO/0012018** (2000), https://arxiv.org/abs/cs/0012018

[25] Hodas, J., Miller, D.: Logic Programming in a Fragment of Intuitionistic Linear Logic. Information and Computation **110**(2), 327 – 365 (1994). https://doi.org/https://doi.org/10.1006/inco.1994.1036

[26] Hughes, J., Marshall, D., Wood, J., Orchard, D.: Linear Exponentials as Graded Modal Types. In: 5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021). Rome (virtual), Italy (Jun 2021), https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465

[27] Hughes, J., Orchard, D.: Resourceful program synthesis from graded linear types. In: Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings. pp. 151–170 (2020). https://doi.org/10.1007/978-3-030-68446-4_8, https://doi.org/10.1007/978-3-030-68446-4_8

[28] Hughes, J., Orchard, D.: Program Synthesis from Graded Types (Additional Material) (Jan 2024). https://doi.org/10.5281/zenodo.10594378, https://zenodo.org/records/10594378

[29] Hughes, J., Vollmer, M., Orchard, D.: Deriving distributive laws for graded linear types. In: Lago, U.D., de Paiva, V. (eds.) Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity&TLLA@IJCAR-FSCD 2020, Online, 29-30 June 2020. EPTCS, vol. 353, pp. 109–131 (2020). https://doi.org/10.4204/EPTCS.353.6, https://doi.org/10.4204/EPTCS.353.6

[30] Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., Sharma, R.: Jigsaw: Large language models meet program synthesis (2021)

[31] Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked borrows: An aliasing model for rust. Proceedings of the ACM on Programming Languages **4**(POPL), 1–32 (2019)

[32] Katsumata, S.: Parametric effect monads and semantics of effect systems. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 633–646. ACM (2014). https://doi.org/10.1145/2535838.2535846

[33] Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers: (functional pearl). SIGPLAN Not. **40**(9), 192–203 (Sep 2005). https://doi.org/10.1145/1090189.1086390

[34] Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. p. 407–426. OOPSLA '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2509136.2509555, https://doi.org/10.1145/2509136.2509555

[35] Knoth, T., Wang, D., Polikarpova, N., Hoffmann, J.: Resource-Guided Program Synthesis. CoRR **abs/1904.07415** (2019), http://arxiv.org/abs/1904.07415

[36] Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. Proceedings of the ACM on Programming Languages **4**(ICFP), 1–29 (2020)

[37] Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**, 90–121 (01 1980). https://doi.org/10.1145/357084.357090

[38] Marshall, D., Vollmer, M., Orchard, D.: Linearity and uniqueness: An entente cordiale. In: Sergey, I. (ed.) Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13240, pp. 346–375. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_13, https://doi.org/10.1007/978-3-030-99336-8_13

[39] McBride, C.: I Got Plenty o' Nuttin', pp. 207–233. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_12

[40] Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3), 348–375 (1978)

[41] Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up synthesis of recursive functional programs using angelic execution. Proceedings of the ACM on Programming Languages **6**(POPL), 1–29 (2022)

[42] Moon, B., III, H.E., Orchard, D.: Graded Modal Dependent Type Theory. In: Yoshida, N. (ed.) Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12648, pp. 462–490. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_17, https://doi.org/10.1007/978-3-030-72019-3_17

[43] de Moura, L., Bjørner, N.: Z3: an efficient smt solver. vol. 4963, pp. 337–340 (04 2008)

[44] Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., Dai, H.: Bustle: Bottom-up program synthesis through learning-guided exploration. arXiv preprint arXiv:2007.14381 (2020)

[45] Orchard, D., Liepelt, V., III, H.E.: Quantitative program reasoning with graded modal types. PACMPL **3**(ICFP), 110:1–110:30 (2019). https://doi.org/10.1145/3341714

[46] Orchard, D.A., Petricek, T., Mycroft, A.: The semantic marriage of monads and effects. CoRR **abs/1401.5391** (2014), http://arxiv.org/abs/1401.5391

[47] Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. SIGPLAN Not. **50**(6), 619–630 (Jun 2015). `https://doi.org/10.1145/2813885.2738007`

[48] Osera, P.M.S.: Program synthesis with types (2015)

[49] Petricek, T., Orchard, D., Mycroft, A.: Coeffects: a calculus of context-dependent computation. In: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming. pp. 123–135. ACM (2014). `https://doi.org/10.1145/2692915.2628160`

[50] Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 522–538. PLDI '16, Association for Computing Machinery, New York, NY, USA (2016). `https://doi.org/10.1145/2908080.2908093`, `https://doi.org/10.1145/2908080.2908093`

[51] Smetsers, S., Barendsen, E., van Eekelen, M., Plasmeijer, R.: Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In: Schneider, H.J., Ehrig, H. (eds.) Graph Transformations in Computer Science. pp. 358–379. Springer Berlin Heidelberg, Berlin, Heidelberg (1994). `https://doi.org/10.1007/3-540-57787-4_23`

[52] Smith, C., Albarghouthi, A.: Synthesizing differentially private programs. Proc. ACM Program. Lang. **3**(ICFP) (Jul 2019). `https://doi.org/10.1145/3341698`

[53] Yuan, Y., Radhakrishna, A., Samanta, R.: Trace-guided inductive synthesis of recursive functional programs. Proceedings of the ACM on Programming Languages **7**(PLDI), 860–883 (2023)

[54] Zalakain, U., Dardha, O.: Pi with leftovers: a mechanisation in Agda. arXiv preprint arXiv:2005.05902 (2020)

Bidirectional Typing and Session Types

A Formal Treatment of Bidirectional Typing

Liang-Ting Chen(✉) and Hsiang-Shang Ko

Institute of Information Science, Academia Sinica, Taipei, Taiwan
{ltchen,joshko}@iis.sinica.edu.tw

Abstract There has been much progress in designing bidirectional type systems and associated type synthesis algorithms, but mainly on a case-by-case basis. To remedy the situation, this paper develops a *general* and *formal* theory of bidirectional typing for simply typed languages: for every signature that specifies a mode-correct bidirectionally typed language, there exists a *proof-relevant* type synthesiser which, given an input abstract syntax tree, constructs a typing derivation if any, gives its refutation if not, or reports that the input does not have enough type annotations. Sufficient conditions for deriving a type synthesiser such as soundness, completeness, and mode-correctness are studied universally for all signatures. We propose a preprocessing step called *mode decoration*, which helps the user to deal with missing type annotations. The entire theory is formally implemented in Agda, so we provide a *verified* generator of proof-relevant type synthesisers as a by-product of our formalism.

1 Introduction

Type inference is an important mechanism for the transition to well-typed programs from untyped abstract syntax trees, which we call *raw terms*. Here 'type inference' refers specifically to algorithms that ascertain the type of any raw term *without type annotations*. However, full parametric polymorphism entails undecidability in type inference, as do dependent types [9, 31]. In light of these limitations, *bidirectional type synthesis* emerged as a viable alternative, deciding the types of raw terms that meet some syntactic criteria and typically contain annotations. In their survey paper [10], Dunfield and Krishnaswami summarised the principles of bidirectional type synthesis and its wide coverage of languages with simple, polymorphic, dependent, and gradual types, among others.

While type inference is not decidable in general, for certain kinds of terms it is still possible to synthesise their types. For example, the type of a variable can be looked up in the context. Bidirectional type synthesis combines type *synthesis* on this subset of terms with type *checking* (based on a given type) on the rest. Formally, every judgement in a bidirectional type system is extended with a *mode:* (i) $\mathit{\Gamma} \vdash t \Rightarrow A$ for *synthesis* and (ii) $\mathit{\Gamma} \vdash t \Leftarrow A$ for *checking*. The former indicates that the type A is an output, using both the context $\mathit{\Gamma}$ and the term t as input, while for the latter, all three of $\mathit{\Gamma}$, t, and A are input. The algorithm of a bidirectional type synthesiser can often be 'read off' from a well-designed bidirectional type system: as the synthesiser traverses a raw term, it switches between synthesis and checking, following the modes assigned to the judgements in the typing rules.

© The Author(s) 2024
S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 115–142, 2024.
https://doi.org/10.1007/978-3-031-57262-3_5

Despite sharing the same basic idea, bidirectional typing has been mostly developed on a case-by-case basis. Dunfield and Krishnaswami present informal design principles learned from individual bidirectional type systems, but in addition to crafting special techniques for individual systems, we should start to consolidate concepts common to a class of bidirectional type systems into a general and formal theory that gives mathematically precise definitions and proves theorems for the class of systems once and for all. In this paper, we develop such a theory of bidirectional typing with the proof assistant AGDA.

Proof-relevant type synthesis Our work adopts a proof-relevant approach to (bidirectional) type synthesis, as illustrated by Wadler et al. [30] for PCF. The proof-relevant formulation deviates from the usual one: traditionally, a type synthesis algorithm is presented as *algorithmic rules*, for example in the form $\Gamma \vdash t \Rightarrow A \mapsto t'$, which denotes that t in the surface language can be transformed to a well-typed term t' of type A in the core language [24]. Such an algorithm is accompanied by soundness and completeness assertions that the algorithm correctly synthesises the type of a raw term, and every typable term can be synthesised. By contrast, the proof-relevant approach exploits the simultaneously computational and logical nature of Martin-Löf type theory, and formulates algorithmic soundness, completeness, and decidability *in one go*.

Recall that the law of excluded middle $P + \neg P$ does not hold as an axiom for every P constructively, and we say that P is logically *decidable* if the law holds for P. Since Martin-Löf type theory is logical and computational, a decidability proof is a proof-relevant decision procedure that computes a yes-or-no answer with a proof of P or its refutation, so logical decidability is algorithmic decidability. More specifically, consider the statement of the type inference problem

> 'for a context Γ and a raw term t, either a typing derivation of $\Gamma \vdash t : A$ exists for some type A or any derivation of $\Gamma \vdash t : A$ for some type A leads to a contradiction',

which can be rephrased more succinctly as

> 'It is <u>decidable</u> for any Γ and t whether $\Gamma \vdash t : A$ is derivable for some A'.

A proof of this statement would also be a program that produces either a typing derivation for the given raw term t or a negation proof that such a derivation is impossible. The first case is algorithmic soundness, while the second case is algorithmic completeness in contrapositive form (which implies the original form due to the decidability). Therefore, proving the statement is the same as constructing a verified proof-relevant type inference algorithm, which returns not only an answer but also a proof justifying that answer. This is an economic way to bridge the gap between theory and practice, where proofs double as verified programs, in contrast to separately exhibiting a theory and an implementation that are loosely related.

Annotations in the type synthesis problem As we mentioned in the beginning, with bidirectional typing we avoid the generally undecidable problem of type inference, and instead solve the simpler problem about the typability of 'sufficiently annotated' raw terms, which we call the type synthesis problem to distinguish it from type inference. Annotations therefore play an important role even in the definition of the problem solved by bidirectional typing, but have not received enough attention. In our theory, we define *mode derivations* to explicitly take annotations into account, and formulate the type synthesis problem with sufficiently annotated raw terms. Accordingly, a preprocessing step called *mode decoration* is proposed to help the user to work with annotations.

The type synthesis problem is not just about deciding whether a raw term is typable—there is a third possibility that the term does not have sufficient annotations. Thus, before attempting to decide typability (using a bidirectional type synthesiser), we should first decide if the raw term has sufficient annotations, which corresponds to whether the term has a mode derivation. Our theory gives a proof-relevant *mode decorator*, which either (i) construct a mode derivation for a raw term, or (ii) provides information that refutes the existence of any mode derivation and pinpoints missing annotations. Then a bidirectional type synthesiser is only required to decide the typability of mode-decorated raw terms. Soundness and completeness of bidirectional typing is reformulated as a one-to-one correspondence between bidirectional typing derivations and pairs of a typing derivation and a mode derivation for the same raw term. Our completeness is simpler and more useful than annotatability, which is a typical formulation of completeness in the literature of bidirectional typing [10, Section 3.2].

Mode-correctness and general definitions of languages The most essential characteristics of bidirectional typing is *mode-correctness*, since an algorithm can often be 'read off' from the definition of a bidirectionally typed language if mode-correct. As illustrated by Dunfield and Krishnaswami [10], it seems that the implications of mode-correctness have only been addressed informally so far, and mode-correctness is not yet formally defined as a *property of languages*.

In order to make the notion of mode-correctness precise, we first give a general definition of bidirectional simple type systems, called *bidirectional binding signature*, extending the typed version of Aczel's binding signature [1] with modes. A general definition of typed languages allows us to define mode-correctness and to investigate its consequences rigorously: the uniqueness of synthesised types and the decidability of bidirectional type synthesis for mode-correct signatures. The proof of the latter theorem amounts to a generator of proof-relevant bidirectional type synthesisers (analogous to a parser generator working for unambiguous or disambiguated grammars).

To make our exposition accessible, the theory in this paper focuses on simply typed languages with a syntax-directed bidirectional type system, so that the decidability of bidirectional type synthesis can be established without any other technical assumptions. It should be possible to extend the theory to deal with more expressive types and assumptions other than mode-correctness. For instance, we briefly discuss how the theory can be extended to handle polymorphically typed languages such as System F, System $\mathsf{F}_{<:}$, and those systems using implicit type applications with additional assumptions in Section 7.

Contributions and plan of this paper In short, we develop a general and formal theory of bidirectional type synthesis for simply typed languages, including

1. general definitions for bidirectional type systems and mode-correctness;
2. mode derivations for explicitly dealing with annotations in the theory, and mode decoration for helping the user to work with annotations in practice;
3. rigorously proven consequences of mode-correctness, including the uniqueness of synthesised types and the decidability of bidirectional type synthesis, which amounts to
4. a fully verified generator of proof-relevant type-synthesisers.

Our theory is fully formally developed with Agda, but is translated to the mathematical vernacular for presentation in this paper. The formal theory doubles as a verified implementation, which is available publicly on Zenodo [8].

This paper is structured as follows. We present a concrete overview of our theory using simply typed λ-calculus in Section 2, prior to developing a general framework for specifying bidirectional type systems in Section 3. Following this, we discuss mode decoration and related properties in Section 4. The main technical contribution lies in Section 5, where we introduce mode-correctness and bidirectional type synthesis. Some examples other than simply typed λ-calculus are given in Section 6, and further developments are discussed in Section 7.

2 Bidirectional type synthesis for simply typed λ-calculus

We start with an overview of our theory by instantiating it to simply typed λ-calculus. Roughly speaking, the problem of type synthesis requires us to take a raw term as input, and produce a typing derivation for the term if possible. To give more precise definitions: the raw terms for simply typed λ-calculus are defined[1] in Figure 1; besides the standard constructs, there is an Anno rule that allows the user to insert type annotations to facilitate type synthesis.

$\boxed{V \vdash t}$ Given a list V of variables, t is a raw term with free variables in V

$$\frac{x \in V}{V \vdash x}\,\textsc{Var} \qquad \frac{V \vdash t}{V \vdash (t \mathbin{⦂} A)}\,\textsc{Anno} \qquad \frac{V, x \vdash t}{V \vdash \lambda x.\, t}\,\textsc{Abs} \qquad \frac{V \vdash t \qquad V \vdash u}{V \vdash t\, u}\,\textsc{App}$$

Figure 1. Raw terms for simply typed λ-calculus

[1] The usual conditions about named representations of variables are omitted.

$\boxed{\Gamma \vdash t : A}$ A raw term t has type A under context Γ

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash (t \mathbin{⨾} A) : A}\ \textsc{Anno} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \supset B}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash t : A \supset B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\; u : B}\ \textsc{App}$$

Figure 2. Typing derivations for simply typed λ-calculus

Correspondingly, the definition of typing derivations[2] in Figure 2 has an Anno rule enforcing that the type of an annotated term does match the annotation.

Now we can define what it means to solve the type synthesis problem.

Definition 2.1. Parametrised by an 'excuse' predicate E on raw terms, a *type synthesiser* takes a context Γ and a raw term $|\Gamma| \vdash t$ (where $|\Gamma|$ is the list of variables in Γ) as input, and establishes one of the following outcomes:

1. there exists a derivation of $\Gamma \vdash t : A$ for some type A,
2. there does not exist a derivation $\Gamma \vdash t : A$ for any type A, or
3. E holds for t.

It is crucial to allow the third outcome, without which we would be requiring the type synthesis problem to be decidable, but this requirement would quickly become impossible to meet when the theory is extended to handle more complex types. If a type synthesiser cannot decide whether there is a typing derivation, it is allowed to give an excuse instead of an answer. Acceptable excuses are defined by the predicate E, which describes what is wrong with an input term, for example, not having enough type annotations.

$\boxed{\Gamma \vdash t \Rightarrow A}$ A raw term t synthesises a type A under Γ

$\boxed{\Gamma \vdash t \Leftarrow A}$ A raw term t checks against a type A under Γ

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{Var}^{\Rightarrow} \qquad \frac{\Gamma \vdash t \Leftarrow A}{\Gamma \vdash (t \mathbin{⨾} A) \Rightarrow A}\ \textsc{Anno}^{\Rightarrow} \qquad \frac{\Gamma \vdash t \Rightarrow B \qquad B = A}{\Gamma \vdash t \Leftarrow A}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{\Gamma, x : A \vdash t \Leftarrow B}{\Gamma \vdash \lambda x.\, t \Leftarrow A \supset B}\ \textsc{Abs}^{\Leftarrow} \qquad \frac{\Gamma \vdash t \Rightarrow A \supset B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\; u \Rightarrow B}\ \textsc{App}^{\Rightarrow}$$

Figure 3. Bidirectional typing derivations for simply typed λ-calculus

[2] We write '$\supset$' instead of '$\to$' for the function types of simply typed λ-calculus to avoid confusion with the function types in our type-theoretic meta-language.

$\boxed{V \vdash t^{\Rightarrow}}$ A raw term t (with free variables in V) is in synthesising mode

$\boxed{V \vdash t^{\Leftarrow}}$ A raw term t (with free variables in V) is in checking mode

$$\frac{x \in V}{V \vdash x^{\Rightarrow}}\ \textsc{Var}^{\Rightarrow} \qquad \frac{V \vdash t^{\Leftarrow}}{V \vdash (t : A)^{\Rightarrow}}\ \textsc{Anno}^{\Rightarrow} \qquad \frac{V \vdash t^{\Rightarrow}}{V \vdash t^{\Leftarrow}}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{V, x \vdash t^{\Leftarrow}}{V \vdash (\lambda x.\, t)^{\Leftarrow}}\ \textsc{Abs}^{\Leftarrow} \qquad \frac{V \vdash t^{\Rightarrow} \qquad V \vdash u^{\Leftarrow}}{V \vdash (t\ u)^{\Rightarrow}}\ \textsc{App}^{\Rightarrow}$$

Figure 4. Mode derivations for simply typed λ-calculus

Now our goal is to use Definition 2.1 as a specification and implement it using a *bidirectional* type synthesiser, which attempts to produce *bidirectional* typing derivations defined in Figure 3. It is often said that a type synthesis algorithm can be 'read off' from well-designed bidirectional typing rules. Take the $\textsc{App}^{\Rightarrow}$ rule as an example: to synthesise the type of an application $t\ u$, we first synthesise the type of t, which should have the form $A \supset B$, from which we can extract the expected type of u, namely A, and perform checking; then the type of the whole application, namely B, can also be extracted from the type $A \supset B$. Note that the synthesiser is able to figure out the type A for checking u and the type B to be synthesised for $t\ u$ because they have been computed when synthesising the type $A \supset B$ of t. In general, there should be a flow of type information in each rule that allows us to determine unknown types (e.g. types to be checked) from known ones (e.g. types previously synthesised). This is called *mode-correctness*, which we will formally define in Section 5.1.

While it is possible for a bidirectional type synthesiser to do its job in one go, which can be thought of as adding both mode and typing information to a raw term and arriving at a bidirectional typing derivation, it is beneficial to have a preprocessing step which adds only mode information, based on which the synthesiser then continues to add typing information. More precisely, the preprocessing step, which we call *mode decoration*, attempts to produce *mode derivations* as defined in Figure 4, where the rules are exactly the mode part of the bidirectional typing rules (Figure 3).

Definition 2.2. A *mode decorator* decides for a raw term $V \vdash t$ whether $V \vdash t^{\Rightarrow}$.

One (less important) benefit of mode decoration is that it helps to simplify the synthesiser, whose computation can be partly directed by a mode derivation. More importantly, whether there is a mode derivation for a term is actually very useful information to the user, because it corresponds to whether the term has enough type annotations: observe that the $\textsc{Anno}^{\Rightarrow}$ and $\textsc{Sub}^{\Leftarrow}$ rules allow us to switch between the synthesising and checking modes; the switch from synthesising to checking is free, whereas the opposite direction requires a type annotation. That is, any term in synthesising mode is also in checking mode, but not necessarily vice versa. A type annotation is required wherever a term that can only be in checking mode is required to be in synthesising mode, and a term does not have a mode derivation if and only if type annotations are missing in such places. (We will treat all these more rigorously in Section 4.) For example, an abstraction is strictly in checking mode, but the left sub-term of an application has to be synthesising, so a term of the form $(\lambda x.\, t)\ u$ does not have a mode derivation unless we annotate the abstraction.

Perhaps most importantly, mode derivations enable us to give bidirectional type synthesisers a tight definition: if we restrict the domain of a synthesiser to terms in synthesising mode (i.e. having enough type annotations for performing synthesis), then it is possible for the synthesiser to *decide* whether there is a suitable typing derivation.

Definition 2.3. A *bidirectional type synthesiser* decides for any context Γ and synthesising term $|\Gamma| \vdash t^{\Rightarrow}$ whether $\Gamma \vdash t \Rightarrow A$ for some type A.

Now we can get back to implementing a type synthesiser (Definition 2.1).

Theorem 2.4. *A type synthesiser using 'not in synthesising mode' as its excuse can be constructed from a mode decorator and a bidirectional type synthesiser.*

The construction is straightforward: run the mode decorator on the input term $|\Gamma| \vdash t$. If there is no synthesising mode derivation, report that t is not in synthesising mode (the third outcome). Otherwise $|\Gamma| \vdash t^{\Rightarrow}$, and we can run the bidirectional type synthesiser. If it finds a derivation of $\Gamma \vdash t \Rightarrow A$ for some type A, return a derivation of $\Gamma \vdash t : A$ (the first outcome), which is possible because the bidirectional typing (Figure 3) is *sound* with respect to the original typing (Figure 2); if there is no derivation of $\Gamma \vdash t \Rightarrow A$ for any type A, then there is no derivation of $\Gamma \vdash t : A$ for any A either (the second outcome), because the bidirectional typing is *complete*:

Theorem 2.5 (Soundness and Completeness). $\Gamma \vdash t \Rightarrow A$ *if and only if* $|\Gamma| \vdash t^{\Rightarrow}$ *and* $\Gamma \vdash t : A$.

We will construct a mode decorator (Section 4.2) and a bidirectional type synthesiser (Section 5) and prove the above theorem for all syntax-directed bidirectional simple type systems (Section 4.1). To quantify over all such systems, we need their general definitions, which we formulate next.

3 Bidirectionally simply typed languages

This section provides general definitions of simple types, simply typed languages, and bidirectional type systems, and uses the simply typed λ-calculus in Section 2 as our running example. These definitions may look dense, especially on first reading. The reader may choose to skim through this section, in particular the figures, and still get some rough ideas from later sections.

The definitions are formulated in two steps: (i) first we introduce a notion of arity and a notion of signature which includes a set[3] of operation symbols and an assignment of arities to symbols; (ii) then, given a signature, we define raw terms and typing derivations inductively by primitive rules such as Var and a rule schema for constructs op_o indexed by an operation symbol o. As we move from simple types to bidirectional typing, the notion of arity, initially as the number of arguments of an operation, is enriched to incorporate an extension context for variable binding and the mode for the direction of type information flow.

3.1 Signatures and simple types

For simple types, the only datum needed for specifying a type construct is its number of arguments:

Definition 3.1. A *signature* Σ for simple types consists of a set I with a decidable equality and an *arity* function $ar \colon I \to \mathbb{N}$. For a signature Σ, a *type* $A : \mathsf{Ty}_\Sigma(\Xi)$ over a variable set Ξ is either

1. a variable in Ξ or
2. $\mathsf{op}_i(A_1, \ldots, A_n)$ for some $i : I$ with $ar(i) = n$ and types $A_1, \ldots, A_n$.

Example 3.2. Function types $A \supset B$ and typically a base type b are included in simply typed λ-calculus, and can be specified by the type signature $\Sigma_\supset$ consisting of operations fun and b where $ar(\mathsf{fun}) = 2$ and $ar(\mathsf{b}) = 0$. Then, all types in simply typed λ-calculus can be given as $\Sigma_\supset$-types over the empty set, with $A \supset B$ introduced as $\mathsf{op}_{\mathsf{fun}}(A, B)$ and b as op_{b}.

Definition 3.3. The *substitution* for a function $\rho \colon \Xi \to \mathsf{Ty}_\Sigma(\Xi')$, denoted by $\rho \colon \mathsf{Sub}_\Sigma(\Xi, \Xi')$, is a map which sends a type $A : \mathsf{Ty}_\Sigma(\Xi)$ to $A\langle\rho\rangle : \mathsf{Ty}_\Sigma(\Xi')$ and is defined as usual.

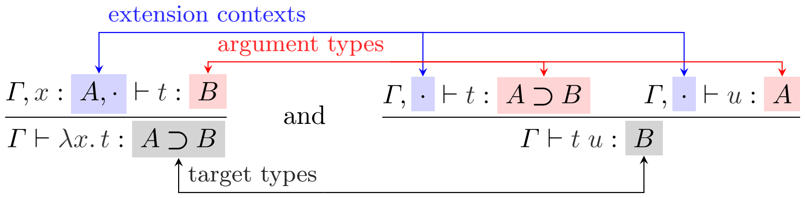
3.2 Binding signatures and simply typed languages

A simply typed language specifies (i) a family of sets of raw terms t indexed by a list V of variables (that are currently in scope), where each construct is allowed to bind some variables like Abs and to take multiple arguments like App; (ii) a family of sets of typing derivations indexed by a typing context Γ, a raw term t, and a type A. Therefore, to specify a term construct, we enrich the notion of arity with some set of types for typing and extension context for variable binding.

Definition 3.4 ([13, p. 322]). A *binding arity* with a set T of types is an inhabitant of $(T^* \times T)^* \times T$, where T^* is the set of lists over T. In a binding arity $(((\Delta_1, A_1), \ldots, (\Delta_n, A_n)), A)$, every Δ_i and A_i refers to the *extension context* and the *type of the i-th argument*, respectively, and A the *target type*. For brevity, it is denoted by $[\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A$, where $[\Delta_i]$ is omitted if empty.

[3] Even though our theory is developed in Martin-Löf type theory, the term 'set' is used instead of 'type' to avoid the obvious confusion. Indeed, as we assume Axiom K, all types are legitimately sets in the sense of homotopy type theory [29, Definition 3.1.1].

Example 3.5. Observe that the Abs and App rules in Figure 2 can be read as



$$\frac{\Gamma, x : A, \cdot \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \supset B} \quad \text{and} \quad \frac{\Gamma, \cdot \vdash t : A \supset B \qquad \Gamma, \cdot \vdash u : A}{\Gamma \vdash t\ u : B}$$

if the empty context $\cdot$ is added verbosely, so they can be specified by arities $[A]B \to (A \supset B)$ and $(A \supset B), A \to B$, respectively, with $\mathsf{Ty}_{\Sigma_\supset}(A, B)$ as types.

Next, akin to a signature, a binding signature Ω consists of a set of operation symbols along with their respective binding arities:

Definition 3.6. For a type signature Σ, a *binding signature* Ω is a set O with a function

$$ar\colon O \to \sum_{\Xi : \mathcal{U}} (\mathsf{Ty}_\Sigma(\Xi)^* \times \mathsf{Ty}_\Sigma(\Xi))^* \times \mathsf{Ty}_\Sigma(\Xi).$$

That is, each inhabitant $o : O$ is associated with a set Ξ of type variables and an arity $ar(o)$ with $\mathsf{Ty}_\Sigma(\Xi)$ as types denoted by $o\colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$.

The set Ξ of type variables for each operation, called its *local context*, plays an important role. To use a rule like Abs in an actual typing derivation, we need to substitute *concrete types*, i.e. types without any type variables, for variables A, B. In our formulation of substitution (3.3), we must first identify which type variables to substitute for. As such, this information forms part of the arity of an operation, and typing derivations, defined subsequently, will include functions ρ from Ξ to concrete types specifying how to instantiate typing rules by substitution.

By a *simply typed language* (Σ, Ω), we mean a pair of a type signature Σ and a binding signature Ω. Now, we define raw terms for (Σ, Ω) first.

Definition 3.7. For a simply typed language (Σ, Ω), the family of sets of *raw terms* indexed by a list V of variables consists of (i) (indices of) variables in V, (ii) annotations $t ⦂ A$ for some raw term t in V and a type A, and (iii) a construct $\mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)$ for some $o\colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ in O, where $\vec{x}_i$'s are lists of variables whose length is equal to the length of Δ_i, and t_i's are raw terms in the variable list $V, \vec{x}_i$. These correspond to rules Var, Anno, and Op in Figure 5 respectively.

Before defining typing derivations, we need a definition of typing contexts.

Definition 3.8. A *typing context* $\Gamma\colon \mathsf{Cxt}_\Sigma$ is formed by $\cdot$ for the empty context and $\Gamma, x : A$ for an additional variable x with a concrete type $A : \mathsf{Ty}_\Sigma(\emptyset)$. The list of variables in Γ is denoted $|\Gamma|$.

The definition of typing derivations is a bit more involved. We need some information to compare types on the object level during type synthesis and substitute those type variables in a typing derivation of $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) : A$ for an operation o in Ω at some point. Here we choose to include a substitution ρ from the local context Ξ to $\emptyset$ as part of its typing derivation explicitly:

$\boxed{V \vdash_{\Sigma,\Omega} t}$ t is a raw term for a language (Σ, Ω) with free variables in V

$$\frac{x \in V}{V \vdash_{\Sigma,\Omega} x}\textsc{Var} \qquad \frac{\cdot \vdash_{\Sigma} A \qquad V \vdash_{\Sigma,\Omega} t}{V \vdash_{\Sigma,\Omega} t \mathbin{⨾} A}\textsc{Anno}$$

$$\frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} t_1 \qquad \cdots \qquad V, \vec{x}_n \vdash_{\Sigma,\Omega} t_n}{V \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)}\textsc{Op}$$

for $o\colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ in Ω

Figure 5. Raw terms

Definition 3.9. For a simply typed language (Σ, Ω), the family of sets of *typing derivations* of $\Gamma \vdash t : A$, indexed by a typing context $\Gamma : \mathsf{Cxt}_{\Sigma}$, a raw term t with free variables in $|\Gamma|$, and a type $A : \mathsf{Ty}_{\Sigma}(\emptyset)$, consists of

1. a derivation of $\Gamma \vdash_{\Sigma,\Omega} x : A$ if $x : A$ is in Γ,
2. a derivation of $\Gamma \vdash_{\Sigma,\Omega} (t \mathbin{⨾} A) : A$ if $\Gamma \vdash_{\Sigma,\Omega} t : A$ has a derivation, and
3. a derivation of $\Gamma \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) : A_0\langle\rho\rangle$ for some operation $o\colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ if there exist $\rho\colon \Xi \to \mathsf{Ty}_{\Sigma}(\emptyset)$ and a derivation of $\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash_{\Sigma,\Omega} t_i : A_i\langle\rho\rangle$ for each i,

corresponding to rules Var, Anno, and Op in Figure 6 respectively.

Example 3.10. Raw terms (Figure 1) and typing derivations (Figure 2) for simply typed λ-calculus can be specified by the type signature $\Sigma_{\supset}$ (Example 3.2) and the binding signature consisting of $\mathsf{app}\colon A, B \rhd (A \supset B), A \to B$ and $\mathsf{abs}\colon A, B \rhd [A]B \to (A \supset B)$. Rules Abs and App in simply typed λ-calculus are subsumed by the Op rule schema, as applications $t\ u$ and abstractions $\lambda x.\, t$ can be introduced uniformly as $\mathsf{op}_{\mathsf{app}}(t, u)$ and $\mathsf{op}_{\mathsf{abs}}(x.t)$, respectively.

3.3 Bidirectional binding signatures and bidirectional type systems

Typing judgements for a bidirectional type system appear in two forms: $\Gamma \vdash t \Rightarrow A$ and $\Gamma \vdash t \Leftarrow A$. These two typing judgements can be considered as a single typing judgement $\Gamma \vdash t :^{d} A$ indexed by a *mode* $d : \mathsf{Mode}$, which can be either $\Rightarrow$ or $\Leftarrow$. Therefore, to define a bidirectional type system, we enrich the concept of binding arity to *bidirectional binding arity*, which further specifies the mode for each of its arguments and for the conclusion:

Definition 3.11. A *bidirectional binding arity* with a set T of types is an inhabitant of

$$(T^* \times T \times \mathsf{Mode})^* \times T \times \mathsf{Mode}.$$

For clarity, an arity is denoted by $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$.

Example 3.12. Consider the $\textsc{Abs}^{\Leftarrow}$ rule (Figure 3) for $\lambda x.\, t$. It has the arity $[A]B^{\Leftarrow} \to (A \supset B)^{\Leftarrow}$, indicating additionally that both $\lambda x.\, t$ and its argument t are checking. Likewise, the $\textsc{App}^{\Rightarrow}$ rule has the arity $(A \supset B)^{\Rightarrow}, A^{\Leftarrow} \to B^{\Rightarrow}$.

Definition 3.13. For a type signature Σ, a *bidirectional binding signature Ω* is a set O with

$$ar \colon O \to \sum_{\Xi : \mathcal{U}} (\mathsf{Ty}_{\Sigma}(\Xi)^{*} \times \mathsf{Ty}_{\Sigma}(\Xi) \times \mathsf{Mode})^{*} \times \mathsf{Ty}_{\Sigma}(\Xi) \times \mathsf{Mode}.$$

We write $o \colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$ for an operation o with a variable set Ξ and its bidirectional binding arity with $\mathsf{Ty}_{\Sigma}(\Xi)$ as types. We call it *checking* if d is $\Leftarrow$ or *synthesising* if d is $\Rightarrow$; similarly, its i-th argument is checking if d_i is $\Leftarrow$ and synthesising if d_i is $\Rightarrow$. A bidirectional type system (Σ, Ω) refers to a pair of a type signature Σ and a bidirectional binding signature Ω.

Definition 3.14. For a bidirectional type system (Σ, Ω),

– the set of *bidirectional typing derivations* of $\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$, indexed by a typing context Γ, a raw term t under $|\Gamma|$, a mode d, and a type A, is defined in Figure 7, and particularly

$$\Gamma \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) :^{d} A_0\langle\rho\rangle$$

has a derivation for $o \colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$ in Ω if there is $\rho \colon \Xi \to \mathsf{Ty}_{\Sigma}(\emptyset)$ and a derivation of $\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash_{\Sigma,\Omega} t_i :^{d_i} A_i\langle\rho\rangle$ for each i;

– the set of *mode derivations* of $V \vdash_{\Sigma,\Omega} t^{d}$, indexed by a list V of variables, a raw term t under V, and a mode d, is defined in Figure 8.

The two judgements $\boxed{\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A}$ and $\boxed{\Gamma \vdash_{\Sigma,\Omega} t \Leftarrow A}$ stand for $\Gamma \vdash_{\Sigma,\Omega} t :^{\Rightarrow} A$ and $\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A$, respectively. A typing rule is *checking* if its conclusion mode is $\Leftarrow$ or *synthesising* otherwise.

Every bidirectional binding signature Ω gives rise to a binding signature $|\Omega|$ if we erase modes from Ω, called the *(mode) erasure* of Ω. Hence a bidirectional type system (Σ, Ω) also specifies a simply typed language $(\Sigma, |\Omega|)$, including raw terms and typing derivations.

Example 3.15. Having established generic definitions, we can now specify simply typed λ-calculus and its bidirectional type system—including raw terms, (bidirectional) typing derivations, and mode derivations—using just a pair of signatures $\Sigma_{\supset}$ (Example 3.2) and $\Omega_{\Lambda}^{\Leftrightarrow}$ which consists of

$$\mathsf{abs} \colon A, B \rhd [A]B^{\Leftarrow} \to (A \supset B)^{\Leftarrow} \quad \text{and} \quad \mathsf{app} \colon A, B \rhd (A \supset B)^{\Rightarrow}, A^{\Leftarrow} \to B^{\Rightarrow}.$$

More importantly, we are able to reason about constructions and properties that hold for any simply typed language with a bidirectional type system once and for all by quantifying over (Σ, Ω).

$\boxed{\Gamma \vdash_{\Sigma,\Omega} t : A}$ t has a concrete type A under Γ for a language (Σ, Ω)

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_{\Sigma,\Omega} x : A}\ \textsc{Var} \qquad\qquad \frac{\Gamma \vdash_{\Sigma,\Omega} t : A}{\Gamma \vdash_{\Sigma,\Omega} (t ⨾ A) : A}\ \textsc{Anno}$$

$$\frac{\rho : \mathsf{Sub}_{\Sigma}(\Xi, \emptyset) \qquad \Gamma, \vec{x}_1 : \Delta_1\langle\rho\rangle \vdash_{\Sigma,\Omega} t_1 : A_1\langle\rho\rangle \ \cdots\ \Gamma, \vec{x}_n : \Delta_n\langle\rho\rangle \vdash_{\Sigma,\Omega} t_n : A_n\langle\rho\rangle}{\Gamma \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) : A_0\langle\rho\rangle}\ \textsc{Op}$$

$$\text{for } o\colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0 \text{ in } \Omega$$

Figure 6. Typing derivations

$\boxed{\Gamma \vdash_{\Sigma,\Omega} t :^{d} A}$ t has a type A in mode d under Γ for a bidirectional system (Σ, Ω)

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_{\Sigma,\Omega} x :^{\Rightarrow} A}\ \textsc{Var}^{\Rightarrow} \qquad\qquad \frac{\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A}{\Gamma \vdash_{\Sigma,\Omega} (t ⨾ A) :^{\Rightarrow} A}\ \textsc{Anno}^{\Rightarrow}$$

$$\frac{\Gamma \vdash_{\Sigma,\Omega} t :^{\Rightarrow} B \qquad B = A}{\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{\begin{array}{c}\rho\colon \mathsf{Sub}_{\Sigma}(\Xi, \emptyset) \\ \Gamma, \vec{x}_1 : \Delta_1\langle\rho\rangle \vdash_{\Sigma,\Omega} t_1 :^{d_1} A_1\langle\rho\rangle \qquad \cdots \qquad \Gamma, \vec{x}_n : \Delta_n\langle\rho\rangle \vdash_{\Sigma,\Omega} t_n :^{d_n} A_n\langle\rho\rangle\end{array}}{\Gamma \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) :^{d} A_0\langle\rho\rangle}\ \textsc{Op}$$

$$\text{for } o\colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d} \text{ in } \Omega$$

Figure 7. Bidirectional typing derivations

$\boxed{V \vdash_{\Sigma,\Omega} t^{d}}$ t is in mode d with free variables in V for (Σ, Ω)

$$\frac{x \in V}{V \vdash_{\Sigma,\Omega} x^{\Rightarrow}}\ \textsc{Var}^{\Rightarrow} \qquad \frac{\cdot \vdash_{\Sigma} A \qquad V \vdash_{\Sigma,\Omega} t^{\Leftarrow}}{V \vdash_{\Sigma,\Omega} (t ⨾ A)^{\Rightarrow}}\ \textsc{Anno}^{\Rightarrow} \qquad \frac{V \vdash_{\Sigma,\Omega} t^{\Rightarrow}}{V \vdash_{\Sigma,\Omega} t^{\Leftarrow}}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} {t_1}^{d_1} \qquad \cdots \qquad V, \vec{x}_n \vdash_{\Sigma,\Omega} {t_n}^{d_n}}{V \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)^{d}}\ \textsc{Op}$$

$$\text{for } o\colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$$

Figure 8. Mode derivations

4 Mode decoration and related properties

Our first important construction is mode decoration in Section 4.2, which is in fact generalised to pinpoint any missing type annotations in a given raw term. We discuss some related properties: by bringing mode derivations into the picture, we are able to give a natural formulation of soundness and completeness of a bidirectional type system with respect to its erasure to an ordinary type system in Section 4.1. We also reformulate annotatability [10, Section 3.2] and compare it with our completeness and generalised mode decoration in Section 4.3.

4.1 Soundness and completeness

Erasure of a bidirectional binding signature removes modes and keeps everything else intact; this can be straightforwardly extended by induction to remove modes from a bidirectional typing derivation and arrive at an ordinary typing derivation, which is soundness. Alternatively, we can remove typing and retain modes, arriving at a mode derivation. Conversely, if we have both mode and typing derivations for the same term, we can combine them to form a bidirectional typing derivation, which is completeness. In short, soundness and completeness are merely the separation and combination of mode and typing information carried by the three kinds of derivations while keeping their basic structure, directed by the same raw term. All these can be summarised in one theorem and proved by induction.

Theorem 4.1. *$\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$ if and only if both $|\Gamma| \vdash_{\Sigma,\Omega} t^{d}$ and $\Gamma \vdash_{\Sigma,|\Omega|} t : A$.*

4.2 Generalised mode decoration

The goal of this section is to construct a mode decorator, which decides for any raw term $V \vdash_{\Sigma,|\Omega|} t$ and mode d whether $V \vdash_{\Sigma,\Omega} t^{d}$ or not. In fact we shall do better: if a mode decorator returns a proof that no mode derivation exists, that negation proof does not give useful information for the user. It will be helpful if a decorator can produce an explanation of why no mode derivation exists, and even how to fix the input term to have a mode derivation. We will construct such a *generalised mode decorator* (Theorem 4.4), which can be weakened to an ordinary mode decorator (Corollary 4.6) if the additional explanation is not needed.[4]

Intuitively, a term does not have a mode derivation exactly when there are not enough type annotations, but such negative formulations convey little information. Instead, we can provide more information by pointing out the places in the term that require annotations. For a bidirectional type system, an annotation is required wherever a term is 'strictly' (which we will define shortly) in checking mode but required to be in synthesising mode, in which case there is no rule for switching from checking to synthesising, and thus there is no way to construct a mode derivation. We can, however, consider *generalised mode derivations* (Figure 9) that allow the use of an additional Missing$^{\Rightarrow}$ rule for such

[4] For the sake of simplicity, we use ordinary mode decoration elsewhere in this paper.

$\boxed{V \vdash_{\Sigma,\Omega} t^{d\,g\,s}}$ t is in mode d, misses some type annotation iff $g = \mathbf{F}$, and is in mode d due to an outermost mode cast iff $s = \mathbf{F}$

$$\frac{x \in V}{V \vdash_{\Sigma,\Omega} x^{\Rightarrow\,\mathbf{T}\,\mathbf{T}}}\ \textsc{Var}^{\Rightarrow} \qquad \frac{\cdot \vdash_{\Sigma} A \qquad V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,s}}{V \vdash_{\Sigma,\Omega} (t \mathbin{⦂} A)^{\Rightarrow\,g\,\mathbf{T}}}\ \textsc{Anno}^{\Rightarrow}$$

$$\frac{V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,\mathbf{T}}}{V \vdash_{\Sigma,\Omega} t^{\Rightarrow\,\mathbf{F}\,\mathbf{F}}}\ \textsc{Missing}^{\Rightarrow} \qquad \frac{V \vdash_{\Sigma,\Omega} t^{\Rightarrow\,g\,\mathbf{T}}}{V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,\mathbf{F}}}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} {t_1}^{d_1\,g_1\,s_1} \qquad \cdots \qquad V, \vec{x}_n \vdash_{\Sigma,\Omega} {t_n}^{d_n\,g_n\,s_n}}{V \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\,t_1; \ldots; \vec{x}_n.\,t_n)^{d\,(\bigwedge_i g_i)\,\mathbf{T}}}\ \textsc{Op}$$

Figure 9. Generalised mode derivations

switching, so that a derivation can always be constructed. Given a generalised mode derivation, if it uses $\textsc{Missing}^{\Rightarrow}$ in some places, then those places are exactly where annotations should be supplied; if it does not use $\textsc{Missing}^{\Rightarrow}$, then the derivation is *genuine* in the sense that it corresponds directly to an original mode derivation. This can be succinctly formulated as Lemma 4.2 below by encoding genuineness as a boolean g in the generalised mode judgement, which is set to $\mathbf{F}$ only by the $\textsc{Missing}^{\Rightarrow}$ rule. (Ignore the boolean s for now.)

Lemma 4.2. *If $V \vdash_{\Sigma,\Omega} t^{d\,\mathbf{T}\,s}$, then $V \vdash_{\Sigma,\Omega} t^{d}$.*

We also want a lemma that covers the case where $g = \mathbf{F}$.

Lemma 4.3. *If $V \vdash_{\Sigma,\Omega} t^{d\,\mathbf{F}\,s}$, then $V \nvdash_{\Sigma,\Omega} t^{d}$.*

This lemma would be wrong if the ‘strictness’ boolean s was left out of the rules: having both $\textsc{Sub}^{\Leftarrow}$ and $\textsc{Missing}^{\Rightarrow}$, which we call *mode casts*, it would be possible to switch between the two modes freely, which unfortunately means that we could insert a pair of $\textsc{Sub}^{\Leftarrow}$ and $\textsc{Missing}^{\Rightarrow}$ anywhere, constructing a non-genuine derivation even when there is in fact a genuine one. The ‘strictness’ boolean s can be thought of as disrupting the formation of such pairs of mode casts: every rule other than the mode casts sets s to $\mathbf{T}$, meaning that a term is *strictly* in the mode assigned by the rule (i.e. not altered by a mode cast), whereas the mode casts set s to $\mathbf{F}$. Furthermore, the sub-derivation of a mode cast has to be strict, so it is impossible to have consecutive mode casts. Another way to understand the role of s is that it makes the $\textsc{Missing}^{\Rightarrow}$ rule precise: an annotation is truly missing only when a term is *strictly* in checking mode but is required to be in synthesising mode. The explicit formulation of strictness makes non-genuine derivations ‘truly non-genuine’, and Lemma 4.3 can be proved.

Now we are ready to construct a generalised mode decorator.

Theorem 4.4 (Generalised mode decoration). *For any raw term $V \vdash_{\Sigma,|\Omega|} t$ and mode d, there is a derivation of $V \vdash_{\Sigma,\Omega} t^{d\,g\,s}$ for some g and s.*

The theorem could be proved directly, but that would mix up two case analyses which respectively inspect the input term t and apply mode casts depending on which mode d is required. Instead, we distill the case analysis on d that deals with mode casts into the following Lemma 4.5, whose antecedent (1) is then established by induction on t in the proof of Theorem 4.4.

Lemma 4.5. *For any raw term $V \vdash_{\Sigma,|\Omega|} t$, if*

$$V \vdash_{\Sigma,\Omega} t^{d'\,g'\,\mathbf{T}} \quad \textit{for some mode } d' \textit{ and boolean } g' \tag{1}$$

then for any mode d, there is a derivation of $V \vdash_{\Sigma,\Omega} t^{d\,g\,s}$ for some g and s.

With a generalised mode decorator, it is now easy to derive an ordinary one.

Corollary 4.6 (Mode decoration). *It is decidable whether $V \vdash_{\Sigma,\Omega} t^{d}$.*

4.3 Annotatability

Dunfield and Krishnaswami [10, Section 3.2] formulated completeness differently from our Theorem 4.1 and proposed *annotatability* as a more suitable name. In our theory, we may reformulate annotatability as follows.

Proposition 4.7 (Annotatability). *If $\Gamma \vdash_{\Sigma,|\Omega|} t : A$, then there exists t' such that $t' \sqsupseteq t$ and $\Gamma \vdash_{\Sigma,\Omega} t' :^{d} A$ for some d.*

Defined in Figure 10, the 'annotation ordering' $t' \sqsupseteq t$ means that t' has the same or more annotations than t. In a sense, annotatability is a reasonable form of completeness: if a term of a simply typed language $(\Sigma, |\Omega|)$ is typable in the ordinary type system, it may not be directly typable in the bidirectional type system (Σ, Ω) due to some missing annotations, but will be if those annotations are added correctly. In our theory, Proposition 4.7 can be straightforwardly proved by induction on the derivation given by generalised mode decoration (Theorem 4.4) to construct a bidirectional typing derivation in the same mode. The interesting case is Missing$^{\Rightarrow}$, which is mapped to Anno$^{\Rightarrow}$, adding to the term a type annotation that comes from the given typing derivation.

$\boxed{t \sqsupseteq u}$ A raw term t is more annotated than u

$$\frac{t \sqsupseteq u}{(t ⦂ A) \sqsupseteq u}\ \textsc{More} \qquad \frac{}{x \sqsupseteq x}\ \textsc{Var} \qquad \frac{t \sqsupseteq u}{(t ⦂ A) \sqsupseteq (u ⦂ A)}\ \textsc{Anno}$$

$$\frac{t_1 \sqsupseteq u_1 \quad \cdots \quad t_n \sqsupseteq u_n}{\mathsf{op}_o(\vec{x}_1.\,t_1; \ldots; \vec{x}_n.\,t_n) \sqsupseteq \mathsf{op}_o(\vec{x}_1.\,u_1; \ldots; \vec{x}_n.\,u_n)}\ \textsc{Op}$$

Figure 10. Annotation ordering between raw terms

On the other hand, when using a bidirectional type synthesiser to implement a type synthesiser, for example in Theorem 2.4, if the bidirectional type synthesiser concludes that there does not exist a bidirectional typing derivation, we use the contrapositive form of completeness to establish that such an ordinary typing derivation does not exist either. Now, annotatability is a kind of completeness because (roughly speaking) it turns an ordinary typing derivation bidirectional. Hence, it is conceivable that we could use annotatability in place of completeness in the proof of Theorem 2.4. However, in the contrapositive form of annotatability, the antecedent is 'there does not exist t' that is more annotated than t and has a bidirectional typing derivation', which is more complex than the bidirectional type synthesiser would have to produce. Annotatability also does not help the user to deal with missing annotations: although annotatability seems capable of determining where annotations are missing and even filling them in correctly, its antecedent requires a typing derivation, which is what the user is trying to construct and does not have yet. Therefore we believe that our theory offers simpler and more useful alternatives than the notion of annotatability.

5 Bidirectional type synthesis and checking

This section focuses on defining mode-correctness and deriving bidirectional type synthesis for any mode-correct bidirectional type system (Σ, Ω). We start with Section 5.1 by defining mode-correctness and showing the uniqueness of synthesised types. This uniqueness means that any two synthesised types for the same raw term t under the same context Γ have to be equal. It will be used especially in Section 5.2 for the proof of the decidability of bidirectional type synthesis and checking. Then, we conclude this section with the trichotomy on raw terms in Section 5.3.

5.1 Mode correctness

As Dunfield and Krishnaswami [10] outlined, mode-correctness for a bidirectional typing rule means that (i) each 'input' type variable in a premise must be an 'output' variable in 'earlier' premises, or provided by the conclusion if the rule is checking; (ii) each 'output' type variable in the conclusion should be some 'output' variable in a premise if the rule is synthesising. Here 'input' variables refer to variables in an extension context and in a checking premise. It is important to note that the order of premises in a bidirectional typing rule also matters, since synthesised type variables are instantiated incrementally during type synthesis.

Consider the rule $\textsc{Abs}^{\Leftarrow}$ (Figure 3) as an example. This rule is mode-correct, as the type variables A and B in its only premise are already provided by its conclusion $A \supset B$. Likewise, the rule $\textsc{App}^{\Rightarrow}$ for an application term $t\ u$ is mode-correct because: (i) the type $A \supset B$ of the first argument t is synthesised, thereby ensuring type variables A and B must be known if successfully synthesised; (ii) the type of the second argument u is checked against A, which has been synthesised earlier; (iii) as a result, the type of an application $t\ u$ can be synthesised.

Now let us define mode-correctness rigorously. As we have outlined, the condition of mode-correctness for a synthesising rule is different from that of a checking rule, and the argument order also matters. Defining the condition directly for a rule, and thus in our setting for an operation, can be somewhat intricate. Instead, we choose to define the conditions for the argument list—more specifically, triples $\overrightarrow{[\Delta_i]A_i^{d_i}}$ of an extension context Δ_i, a type A_i, and a mode d_i—pertaining to an operation, for an operation, and subsequently for a signature. We also need some auxiliary definitions for the subset of variables of a type and of an extension context, and the set of variables that have been synthesised:

Definition 5.1. The finite subsets[5] of *(free) variables* of a type A and of variables in an extension context Δ are denoted by $\mathit{fv}(A)$ and $\mathit{fv}(\Delta)$ respectively. For an argument list $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$, the set of type variables $A_i^{d_i}$ with d_i being $\Rightarrow$ is denoted by $\mathit{fv}^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$, i.e. $\mathit{fv}^{\Rightarrow}$ gives the set of type variables that will be synthesised during type synthesis.

Definition 5.2. The *mode-correctness* $\mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right)$ for an argument list $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$ with respect to a subset S of Ξ is a predicate defined by

$$\mathsf{MC}_{as}(\cdot) = \top$$

$$\mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}, [\Delta_n]A_n^{\Leftarrow}\right) = \mathit{fv}(\Delta_n, A_n) \subseteq \left(S \cup \mathit{fv}^{\Rightarrow}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right)\right) \wedge \mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right)$$

$$\mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}, [\Delta_n]A_n^{\Rightarrow}\right) = \mathit{fv}(\Delta_n) \subseteq \left(S \cup \mathit{fv}^{\Rightarrow}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right)\right) \wedge \mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right)$$

where $\mathsf{MC}_{as}(\cdot) = \top$ means that an empty list is always mode-correct.

This definition encapsulates the idea that every 'input' type variable, possibly derived from an extension context Δ_n or a checking argument A_n, must be an 'output' variable from $\mathit{fv}^{\Rightarrow}(\overrightarrow{[\Delta_i]A_i^{d_i}})$ or, if the rule is checking, belong to the set S of 'input' variables in its conclusion. This condition must also be met for every tail of the argument list to ensure that 'output' variables accessible at each argument are from preceding arguments only, hence an inductive definition.

Definition 5.3. An arity $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^d$ is *mode-correct* if

1. either d is $\Leftarrow$, its argument list is mode-correct with respect to $\mathit{fv}(A_0)$, and the union $\mathit{fv}(A_0) \cup \mathit{fv}^{\Rightarrow}(\overrightarrow{[\Delta_i]A_i^{d_i}})$ contains every inhabitant of Ξ;
2. or d is $\Rightarrow$, its argument list is mode-correct with respect to $\emptyset$, and $\mathit{fv}^{\Rightarrow}(\overrightarrow{[\Delta_i]A_i^{d_i}})$ contains every inhabitant of Ξ and, particularly, $\mathit{fv}(A_0)$.

A bidirectional binding signature Ω is *mode-correct* if every operation's arity is mode-correct.

[5] There are various definitions for finite subsets of a set within type theory, but for our purposes the choice among these definitions is not a matter of concern.

For a checking operation, an 'input' variable of an argument could be derived from A_0, as these are known during type checking as an input. Since every inhabitant of Ξ can be located in either A_0 or synthesised variables, we can determine a concrete type for each inhabitant of Ξ during type synthesis. On the other hand, for a synthesising operation, we do not have any known variables at the onset of type synthesis, so the argument list should be mode-correct with respect to $\emptyset$. Also, the set of synthesised variables alone should include every type variable in Ξ and particularly in A_n.

Remark 5.4. Mode-correctness is fundamentally a condition for bidirectional typing *rules*, not for derivations. Thus, this property cannot be formulated without treating rules as some mathematical object such as those general definitions in Section 3. This contrasts with the properties in Section 4, which can still be specified for individual systems in the absence of a general definition.

It is easy to check the bidirectional type system $(\Sigma_{\supset}, \Omega_{\Lambda}^{\Leftrightarrow})$ for simply typed λ-calculus is mode-correct by definition or by the following lemma:

Lemma 5.5. *For any bidirectional binding arity* $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$*, it is decidable whether it is mode-correct.*

Now, we set out to show the uniqueness of synthesised types for a mode-correct bidirectional type system. For a specific system, its proof is typically a straightforward induction on the typing derivations. However, since mode-correctness is inductively defined on the argument list, our proof proceeds by induction on both the typing derivations and the argument list:

Lemma 5.6 (Uniqueness of synthesised types). *In a mode-correct bidirectional type system* (Σ, Ω)*, the synthesised types of any two derivations*

$$\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A \quad \textit{and} \quad \Gamma \vdash_{\Sigma,\Omega} t \Rightarrow B$$

for the same term t *must be equal, i.e.* $A = B$.

Proof. We prove the statement by induction on derivations d_1 and d_2 for $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A$ and $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow B$. Our system is syntax-directed, so d_1 and d_2 must be derived from the same rule:

– Var$^{\Rightarrow}$ follows from the fact that each variable as a raw term refers to the same variable in its context.
– Anno$^{\Rightarrow}$ holds trivially, since the synthesised type A is from the term $t \mathbin{⦂} A$ in question.
– Op: Recall that a derivation of $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A$ contains a substitution ρ from the local context Ξ to concrete types. To prove that any two typing derivations has the same synthesised type, it suffices to show that those substitutions ρ_1 and ρ_2 of d_1 and d_2, respectively, agree on variables in $fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$ so that $A_0\langle\rho_1\rangle = A_0\langle\rho_2\rangle$. We prove it by induction on the argument list:
  1. For the empty list, the statement is vacuously true.

2. If d_{i+1} is $\Leftarrow$, then the statement holds by induction hypothesis.
3. If d_{i+1} is $\Rightarrow$, then $\Delta_{i+1}\langle\rho_1\rangle = \Delta_{i+1}\langle\rho_2\rangle$ by induction hypothesis (of the list). Therefore, under the same context $\Gamma, \Delta_{i+1}\langle\rho_1\rangle = \Gamma, \Delta_{i+1}\langle\rho_2\rangle$ the term t_{i+1} must have the same synthesised type $A_{i+1}\langle\rho_1\rangle = A_{i+1}\langle\rho_1\rangle$ by induction hypothesis (of the typing derivation), so ρ_1 and ρ_2 agree on $\mathit{fv}(A_{i+1})$ in addition to $\mathit{fv}^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$.

$\square$

5.2 Decidability of bidirectional type synthesis and checking

We have arrived at the main technical contribution of this paper.

Theorem 5.7. *In a mode-correct bidirectional type system (Σ, Ω),*

1. *if $|\Gamma| \vdash_{\Sigma,\Omega} t^{\Rightarrow}$, then it is decidable whether $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A$ for some A;*
2. *if $|\Gamma| \vdash_{\Sigma,\Omega} t^{\Leftarrow}$, then it is decidable for any A whether $\Gamma \vdash_{\Sigma,\Omega} t \Leftarrow A$.*

The interesting part of the theorem is the case for the Op rule. We shall give its insight first instead of jumping into the details. Recall that a typing derivation for $\mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)$ contains a substitution $\rho\colon \Xi \to \mathsf{Ty}_\Sigma(\emptyset)$. The goal of type synthesis for this case is exactly to define such a substitution ρ, and we have to start with an 'accumulating' substitution: a substitution ρ_0 that is partially defined on $\mathit{fv}(A_0)$ if d is $\Leftarrow$ or otherwise nowhere. By mode-correctness, the accumulating substitution ρ_i will be defined on enough synthesised variables so that type synthesis or checking can be performed on t_i with the context $\Gamma, \vec{x}_i : \Delta_i\langle\rho_i\rangle$ based on its mode derivation $|\Gamma|, \vec{x}_i \vdash_{\Sigma,\Omega} t_i^{d_i}$. If we visit a synthesising argument $[\Delta_{i+1}]A_{i+1}^{\Rightarrow}$, then we may extend the domain of ρ_i to ρ_{i+1} with the synthesised variables $\mathit{fv}(A_{i+1})$, provided that type synthesis is successful and that the synthesised type can be *unified with* A_{i+1}. If we go through every t_i successfully, then we will have a total substitution ρ_n by mode-correctness and a derivation of $\Gamma, \vec{x}_i : \Delta_i \vdash_{\Sigma,\Omega} t_i :^{d_i} A\langle\rho_n\rangle$ for each sub-term t_i.

Remark 5.8. To make the argument above sound, it is necessary to compare types and solve a unification problem. Hence, we assume that the set Ξ of type variables has a decidable equality, thereby ensuring that the set $\mathsf{Ty}_\Sigma(\Xi)$ of types also has a decidable equality.[6]

We need some auxiliary definitions for the notion of extension to state the unification problem:

Definition 5.9. By an *extension* $\sigma \geq \rho$ of a partial substitution ρ we mean that the domain $\mathit{dom}(\sigma)$ of σ contains the domain of ρ and $\sigma(x) = \rho(x)$ for every x in $\mathit{dom}(\rho)$. By a *minimal extension* $\bar{\rho}$ of ρ satisfying P we mean an extension $\bar{\rho} \geq \rho$ with $P(\bar{\rho})$ such that $\sigma \geq \bar{\rho}$ whenever $\sigma \geq \rho$ and $P(\sigma)$.

[6] To simplify our choice, we may confine Ξ to any set within the family of sets $\mathsf{Fin}(n)$ of naturals less than n, given that these sets have a decidable equality and the arity of a type construct is finite. Indeed, in our formalisation, we adopt $\mathsf{Fin}(n)$ as the set of type variables in the definition of Ty_Σ. For the sake of clarity in presentation, though, we just use named variables and assume that Ξ has a decidable equality.

Lemma 5.10. *For any A of $\mathsf{Ty}_\Sigma(\Xi)$, B of $\mathsf{Ty}_\Sigma(\emptyset)$, and a partial substitution $\rho\colon \Xi \to \mathsf{Ty}_\Sigma(\emptyset)$, either*

1. *there is a minimal extension $\bar\rho$ of ρ such that $A\langle\bar\rho\rangle = B$, or*
2. *there is no extension σ of ρ such that $A\langle\sigma\rangle = B$*

This lemma can be derived from the correctness of first-order unification [21, 22], or be proved directly without unification. We are now ready for Theorem 5.7:

Proof (of Theorem 5.7). We prove this statement by induction on the mode derivation $|\Gamma| \vdash_{\Sigma,\Omega} t^d$. The two cases $\textsc{Var}^{\Rightarrow}$ and $\textsc{Anno}^{\Rightarrow}$ are straightforward and independent of mode-correctness. The case $\textsc{Sub}^{\Leftarrow}$ invokes the uniqueness of synthesised types to refute the case that $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow B$ but $A \neq B$ for a given type A. The first three cases follow essentially the same reasoning provided by Wadler et al. [30], so we only detail the last case $\textsc{Op}$, which is new (but has been discussed informally above). For brevity we omit the subscript (Σ, Ω).

For a mode derivation of $|\Gamma| \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)^d$, we first claim:

Claim. For an argument list $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$ and any partial substitution ρ from Ξ to $\emptyset$, either

1. there is a minimal extension $\bar\rho$ of ρ such that
$$dom(\bar\rho) \supseteq fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}) \text{ and } \Gamma, \vec{x}_i : \Delta_i\langle\bar\rho\rangle \vdash t_i \colon A_i\langle\bar\rho\rangle^{d_i} \qquad (2)$$
for $i = 1, \ldots, n$, or
2. there is no extension σ of ρ such that (2) holds.

Then, we proceed with a case analysis on d in the mode derivation:

– d is $\Rightarrow$: We apply our claim with the partial substitution ρ_0 defined nowhere.
  1. If there is no $\sigma \geq \rho$ such that (2) holds but $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A$ for some A, then by inversion we have $\rho\colon \mathsf{Sub}_\Sigma(\Xi, \emptyset)$ such that
  $$\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash t_i \colon A_i\langle\rho\rangle^{d_i}$$
  for every i. Obviously, $\rho \geq \rho_0$ and $\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash t_i \colon A_i\langle\rho\rangle^{d_i}$ for every i, which contradict the assumption that no such extension exists.
  2. If there exists a minimal $\bar\rho \geq \rho_0$ defined on $fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$ such that (2) holds, then by mode-correctness $\bar\rho$ is total, and thus
  $$\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A_0\langle\bar\rho\rangle\,.$$

– d is $\Leftarrow$: Let A be a type and apply Lemma 5.10 with ρ_0 defined nowhere.
  1. If there is no $\sigma \geq \rho_0$ s.t. $A_0\langle\sigma\rangle = A$ but $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A$, then inversion gives us a substitution ρ s.t. $A = A_0\langle\rho\rangle$—a contradiction.
  2. If there is a minimal $\bar\rho \geq \rho_0$ s.t. $A_0\langle\bar\rho\rangle = A$, then apply our claim with $\bar\rho$:

(a) If no $\sigma \geq \bar{\rho}$ satisfies (2) but $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A$, then by inversion there is γ s.t. $A_0\langle\gamma\rangle = A$ and $\Gamma, \vec{x}_i : \Delta_i\langle\gamma\rangle \vdash t_i \colon A_i\langle\gamma\rangle^{d_i}$ for every i. Given that $\bar{\rho} \geq \rho$ is minimal s.t. $A_0\langle\bar{\rho}\rangle = A$, it follows that γ is an extension of $\bar{\rho}$, but by assumption no such an extension satisfying $\Gamma, \vec{x}_i : \Delta_i\langle\gamma\rangle \vdash t_i \colon A_i\langle\gamma\rangle^{d_i}$ exists, thus a contradiction.

(b) If there is a minimal $\bar{\bar{\rho}} \geq \bar{\rho}$ s.t. (2), then by mode-correctness $\bar{\bar{\rho}}$ is total and

$$\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A_0\langle\bar{\bar{\rho}}\rangle$$

where $A_0\langle\bar{\bar{\rho}}\rangle = A_0\langle\bar{\rho}\rangle = A$ since $\bar{\bar{\rho}}(x) = \bar{\rho}$ for every x in $dom(\bar{\rho})$.

We have proved the decidability by induction on the derivation of $|\Gamma| \vdash_{\Sigma,\Omega} t^d$, assuming the claim.

Proof (of Claim). We prove it by induction on the list $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$:

1. For the empty list, ρ is the minimal extension of ρ itself satisfying (2) trivially.
2. For $\overrightarrow{[\Delta_i]A_i^{d_i}}, [\Delta_{m+1}]A_{m+1}^{d_{m+1}}$, by induction hypothesis on the list, we have two cases:
   (a) If there is no $\sigma \geq \rho$ s.t. (2) holds for all $1 \leq i \leq m$ but a minimal $\gamma \geq \rho$ such that (2) holds for all $1 \leq i \leq m+1$, then we have a contradiction.
   (b) There is a minimal $\bar{\rho} \geq \rho$ s.t. (2) holds for $1 \leq i \leq m$. By case analysis on d_{m+1}:
      – d_{m+1} is $\Leftarrow$: By mode-correctness, $\Delta_{m+1}\langle\bar{\rho}\rangle$ and $A_{m+1}\langle\bar{\rho}\rangle$ are defined. By the ind. hyp. $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Leftarrow A_{m+1}\langle\bar{\rho}\rangle$ is decidable. Clearly, if $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Leftarrow A_{m+1}\langle\bar{\rho}\rangle$ then the desired statement is proved; otherwise we easily derive a contradiction.
      – d_{m+1} is $\Rightarrow$: By mode-correctness, $\Delta_{m+1}\langle\bar{\rho}\rangle$ is defined. By the ind. hyp., '$\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A$ for some A' is decidable:
         i. If $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \nvdash t_{m+1} \Rightarrow A$ for any A but there is $\gamma \geq \rho$ s.t. (2) holds for $1 \leq i \leq m+1$, then $\gamma \geq \bar{\rho}$. Therefore $\Delta_{m+1}\langle\bar{\rho}\rangle = \Delta_{m+1}\langle\gamma\rangle$, and we derive a contradiction because $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A_{m+1}\langle\gamma\rangle$.
         ii. If $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A$ for some A, then Lemma 5.10 gives the following two cases:
            • Suppose no $\sigma \geq \bar{\rho}$ s.t. $A_{m+1}\langle\sigma\rangle = A$ but an extension $\gamma \geq \rho$ s.t. (2) holds for $1 \leq i \leq m+1$. Then, $\gamma \geq \bar{\rho}$ by the minimality of $\bar{\rho}$ and thus $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A_{m+1}\langle\gamma\rangle$. However, by Lemma 5.6, the synthesised type $A_{m+1}\langle\gamma\rangle$ must be unique, so γ is an extension of $\bar{\rho}$ s.t. $A_{m+1}\langle\gamma\rangle = A$, i.e. a contradiction.
            • If there is a minimal $\bar{\bar{\rho}} \geq \bar{\rho}$ such that $A_{m+1}\langle\bar{\bar{\rho}}\rangle = A$, then it is not hard to show that $\bar{\bar{\rho}}$ is also the minimal extension of ρ such that (2) holds for all $1 \leq i \leq m+1$.

We have proved our claim for any argument list by induction. ■

We have completed the proof of Theorem 5.7. □

The formal counterpart of the above proof in AGDA functions as two top-level programs for type checking and synthesis. These programs provide either the typing derivation or its negation proof. Each case analysis branches depending on the outcomes of bidirectional type synthesis and checking for each sub-term, as well as the unification process. If a negation proof is not of interest in practice, these programs can be simplified by discarding the cases that yield negation proofs. Alternatively, we could consider generalising typing derivations instead, like our generalised mode derivations (Figure 9), to reformulate negation proofs positively to deliver more informative error messages. This would assist programmers in resolving issues with ill-typed terms, rather than returning a blatant 'no'.

5.3 Trichotomy on raw terms by type synthesis

Combining the bidirectional type synthesiser with the mode decorator, soundness, and completeness from Section 4, we derive a type synthesiser parameterised by (Σ, Ω), generalising Theorem 2.4.

Corollary 5.11 (Trichotomy on raw terms). *For any mode-correct bidirectional type system (Σ, Ω), exactly one of the following holds:*

1. $|\Gamma| \vdash_{\Sigma,\Omega} t^{\Rightarrow}$ *and* $\Gamma \vdash_{\Sigma,|\Omega|} t : A$ *for some type* A,
2. $|\Gamma| \vdash_{\Sigma,\Omega} t^{\Rightarrow}$ *but* $\Gamma \nvdash_{\Sigma,|\Omega|} t : A$ *for any type* A, *or*
3. $|\Gamma| \nvdash_{\Sigma,\Omega} t^{\Rightarrow}$.

6 Examples

To exhibit the applicability of our approach, we discuss two more examples: one has infinitely many operations and the other includes many more constructs than simply typed λ-calculus, exhibiting the practical side of a general treatment.

6.1 Spine application

A spine application $t\ u_1\ \ldots\ u_n$ is a form of application that consists of a head term t and an indeterminate number of arguments u_1, u_2, …, u_n. This arrangement allows direct access to the head term, making it practical in various applications, and has been used by AGDA's core language.

At first glance, accommodating this form of application may seem impossible, given that the number of arguments for a construct is finite and has to be fixed. Nonetheless, the total number of operation symbols in a signature need not be finite, allowing us to establish a corresponding construct for each number n of arguments, i.e. viewing the following rule

$$\frac{\Gamma \vdash t \Rightarrow A_1 \supset (A_2 \supset (\cdots \supset (A_n \supset B)\ldots)) \qquad \Gamma \vdash u_1 \Leftarrow A_1 \qquad \cdots \qquad \Gamma \vdash u_n \Leftarrow A_n}{\Gamma \vdash t\, u_1\ \ldots\ u_n \Rightarrow B}$$

as a rule schema parametrised by n, so the signature $\Omega^{\Leftrightarrow}_{\Lambda}$ can be extended with

$$\mathsf{app}_n\colon A_1, \ldots, A_n, B \rhd A_1 \supset (A_2 \supset (\cdots \supset (A_n \supset B)\ldots))^{\Rightarrow}, A_1^{\Leftarrow}, \ldots, A_n^{\Leftarrow} \to B$$

Each application $t\, u_1\ \ldots\ u_n$ can be introduced as $\mathsf{op}_{\mathsf{app}_n}(t; u_1; \ldots; u_n)$, thereby exhibiting the necessity of having an arbitrary set for operation symbols.

6.2 Computational calculi

Implementing a stand-alone type synthesiser for a simply typed language is typically a straightforward task. However, the code size increases proportionally to the number of type constructs and of arguments associated with each term construct. For example, when dealing with a fixed number n of type constructs, for each synthesising construct there are two cases for a checking argument but $n+1$ cases for each synthesising argument: the successful synthesis of the expected type, an instance where it fails, or $n-1$ cases where the expected type does not match. Thus, having a generator is helpful and can significantly reduce the effort for implementation.

For illustrative purposes, consider a computational calculus [23] with naturals, sums, products, and general recursion. The extended language has 'only' 15 constructs, including pairing, projections, injections, and so on, and this number of constructs is still far fewer than what a realistic programming language would have. Even for this small calculus, there are already nearly 100 possible cases to consider in bidirectional type synthesis.

On the other hand, similar to a parser generator, only one specification is needed for a type-synthesiser generator from the user to produce a corresponding type synthesiser. Such a specification can be derived by extending $(\Sigma_{\supset}, \Omega_{\Lambda}^{\Leftrightarrow})$ accordingly for additional types and constructs with mode-correctness proved by applying Lemma 5.5, so its type synthesiser follows from Corollary 5.11 directly.

7 Discussion

We believe that our formal treatment lays a foundation for further investigation, as the essential aspects of bidirectional typing have been studied rigorously. While our current development is based on simply typed languages to highlight the core ideas, it is evident that many concepts and aspects remain untouched.

Language formalisation frameworks The idea of presenting logics universally at least date back to universal algebra and model theory, where structures are studied for certain notions of arities and signatures. In programming language theory, Aczel's binding signature [1] is an example which has been used to prove a general confluence theorem. Many general definitions and frameworks for defining logics and type theories have been proposed and can be classified into two groups by where signatures reside—the meta level or the object level of a meta-language:

1. Harper et al.'s logical framework LF [17] and its family of variants [5, 11, 18, 25] are *extensions* of Martin-Löf type theory, where signatures are on the *meta level* and naturally capable of specifying dependent type theories;
2. general dependent type theories [6, 7, 19, 28], categorical semantics [4, 12–16, 26, 27] (which includes the syntactic model as a special case), and frameworks for substructural systems [26, 27, 32] are developed *within* a meta-theory (set theory or type theory), where signatures are on the *object level* and their expressiveness varies depending on their target languages.

The LF family is expressive, but each extension is a different metalanguage and requires a different implementation to check formal LF proofs. Formalising LF and its variants is at least as complicated as formalising a dependent type theory, and they are mostly implemented separately from their theory and unverified.

For the second group, theories developed in set theory can often be restated in type theory and thus manageable for formalisation in a type-theoretic proof assistant. Such examples include frameworks developed by Ahrens et al. [2], Allais et al. [3], Fiore and Szamozvancev [14], although these formal implementations are limited to simply typed theories for now.

Our work belongs to the second group, as we aim for a formalism in a type theory to minimise the gap between theory and implementation.

Beyond simple types Bidirectional type synthesis plays a crucial role in handling more complex types than simple types, and we sketch how our theory can be extended to treat a broader class of languages. First, we need a general definition of languages in question (Sections 3.1 and 3.2). Then, this definition can be augmented with modes (Section 3.3) and the definition of mode-correctness (Definition 5.3) can be adapted accordingly. Soundness and completeness (Theorem 4.1) should still hold, as they amount to the separation and combination of mode and typing information for a given raw term (in a syntax-directed formulation). Mode decoration (Section 4.2), which annotates a raw term with modes and marks missing annotations, should also work. As for the decidability of bidirectional type synthesis, we discuss two cases involving polymorphic types and dependent types below.

Polymorphic types In the case of languages like System F and others that permit type-level variable binding, we can start with the notion of polymorphic signature, as introduced by Hamana [16]—(i) each type construct in a signature is specified by a binding arity with only one type $*$, and (ii) a term construct can employ a pair of extension contexts for term variables and type variables.

Extending general definitions for bidirectional typing and mode derivations from Hamana's work is straightforward. For example, the universal type $\forall\alpha.\,A$ and type abstraction in System F can be specified as operations $\mathsf{all} : * \rhd [*]* \to *$ and $\mathsf{tabs} : [*]A \rhd \langle * \rangle\, A^{\Leftarrow} \to \mathsf{op}_{\mathsf{all}}(\alpha.A)^{\Leftarrow}$. The decidability of bidirectional type synthesis (Theorem 5.7) should also carry over, as no equations are imposed on types and no guessing (for type application) is required. Adding subtyping $A <: B$ to languages can be done by replacing type equality with a subtyping relation $<:$ and type equality check with subtyping check, so polymorphically typed languages with subtyping such as System $\mathsf{F}_{<:}$ can be specified. The main idea of bidirectional typing does not change, so it should be possible to extend the formal theory without further assumptions too.

However, explicit type application in System F and System $\mathsf{F}_{<:}$ is impractical but its implicit version results in a *stationary rule* [20] which is not syntax-directed. By translating the rule to subtyping, we have the *instantiation problem* that requires guessing B in $\forall\alpha.A <: A[B/\alpha]$. A theory that accommodates various solutions to the problem is left as future work.

Dependent types Logical frameworks with bidirectional typing are proposed by Reed [25] and Felicissimo [11]. Felicissimo's framework is more expressive than Reed's, due to its ability to specify rewriting rules. Both frameworks extend LF, enabling generic bidirectional type checking for dependent type theories. They also incorporate notions of signatures and mode-correctness (called *strictness* and *validity*, respectively, in their contexts) but differ from ours in several ways.

First, the number of operations introduced by a signature in LF is finite, so constructs like spine application seem impossible to define. Second, Reed and Felicissimo deal with decorated raw terms only, while our theory bridges the gap between ordinary and decorated raw terms by mode decoration. Lastly, Felicissimo classifies operations *a priori* into introduction and elimination rules, and follows the Pfenning recipe assigning, for example, the synthesising mode to each elimination rule and its principal argument. As pointed out by Dunfield and Krishnaswami that bidirectional typing is essentially about managing information flow, and that some systems remarkably deviate from this recipe, we do not enforce it but establish our results on any reasonable information flow.

Beyond syntax-directedness To relax the assumption of syntax-directedness, we could start from a simple but common case where the ordinary typing part is still syntax-directed, but each typing rule is refined to multiple bidirectional variants, including different orders of its premises. In such cases, the mode decorator would need to backtrack and find all mode derivations, but the type synthesiser should still work in a syntax-directed manner on each mode derivation. Completeness could still take the simple form presented in this paper too.

Next, we could consider systems where each construct can have multiple typing rules, which can further have multiple bidirectional variants. In this setting, the bidirectional type synthesiser will also need to backtrack. It is still possible to treat soundness as the separation of mode and type information, but completeness will pose a problem: for every raw term, a mode derivation chooses a mode assignment while a typing derivation chooses a typing rule, but there may not be a bidirectional typing rule for this particular combination. A solution might be refining completeness to say that any typing derivation can be combined with one of the possible mode derivations into a bidirectional typing derivation.

Towards a richer formal theory There are more principles and techniques in bidirectional typing that could be formally studied in general, with one notable example being the Pfenning recipe for bidirectionalising typing rules [10, Section 4]. There are also concepts that may be hard to fully formalise, for example 'annotation character' [10, Section 3.4], which is roughly about how easy it is for the user to write annotated programs, but it would be interesting to explore to what extent such concepts can be formalised.

Acknowledgements. We thank Kuen-Bang Hou (Favonia) and anonymous reviewers for their comments and suggestions. The work was supported by the National Science and Technology Council of Taiwan under grant NSTC 112-2221-E-001-003-MY3.

Bibliography

[1] Aczel, P.: A general Church–Rosser theorem (1978), URL http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf

[2] Ahrens, B., Matthes, R., Mörtberg, A.: Implementing a category-theoretic framework for typed abstract syntax. In: International Conference on Certified Programs and Proofs (CPP), pp. 307–323, ACM (2022), https://doi.org/10.1145/3497775.3503678

[3] Allais, G., Atkey, R., Chapman, J., McBride, C., McKinna, J.: A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. Journal of Functional Programming **31**, e22:1–55 (2021), https://doi.org/10.1017/S0956796820000076

[4] Arkor, N., Fiore, M.: Algebraic models of simple type theories: a polynomial approach. In: Symposium on Logic in Computer Science (LICS), pp. 88–101, ACM (2020), https://doi.org/10.1145/3373718.3394771

[5] Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\Pi$-Calculus Modulo Theory (2023), https://doi.org/10.48550/arXiv.2311.07185

[6] Bauer, A., Haselwarter, P.G., Lumsdaine, P.L.: A general definition of dependent type theories (2020), https://doi.org/10.48550/arXiv.2009.05539

[7] Bauer, A., Komel, A.P.: An extensible equality checking algorithm for dependent type theories. Logical Methods in Computer Science **18**(1), 17:1–42 (2022), https://doi.org/10.46298/lmcs-18(1:17)2022

[8] Chen, L.T., Ko, H.S.: A formal treatment of bidirectional typing (artefact) (2024), https://doi.org/10.5281/zenodo.10458840

[9] Dowek, G.: The undecidability of typability in the lambda-pi-calculus. In: International Conference on Typed Lambda Calculi and Applications (TLCA), Lecture Notes in Computer Science, vol. 664, pp. 139–145, Springer (1993), https://doi.org/10.1007/BFb0037103

[10] Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Computing Surveys **54**(5), 98:1–38 (2021), https://doi.org/10.1145/3450952

[11] Felicissimo, T.: Generic bidirectional typing for dependent type theories (2023), https://doi.org/10.48550/arXiv.2307.08523

[12] Fiore, M., Hamana, M.: Multiversal polymorphic algebraic theories: syntax, semantics, translations, and equational logic. In: Symposium on Logic in Computer Science (LICS), pp. 520–529, IEEE (2013), https://doi.org/10.1109/LICS.2013.59

[13] Fiore, M., Hur, C.K.: Second-order equational logic (extended abstract). In: International Workshop on Computer Science Logic (CSL), Lecture Notes in Computer Science, vol. 6247, pp. 320–335, Springer (2010), https://doi.org/10.1007/978-3-642-15205-4_26

[14] Fiore, M., Szamozvancev, D.: Formal metatheory of second-order abstract syntax. Proceedings of the ACM on Programming Languages **6**(POPL), 53:1–29 (2022), https://doi.org/10.1145/3498715

[15] Fiore, M.P., Plotkin, G.D., Turi, D.: Abstract syntax and variable binding. In: Symposium on Logic in Computer Science (LICS), pp. 193–202, IEEE (1999), https://doi.org/10.1109/LICS.1999.782615

[16] Hamana, M.: Polymorphic abstract syntax via Grothendieck construction. In: International Conference on Foundations of Software Science and Computational Structures (FoSSaCS), Lecture Notes in Computer Science, vol. 6604, pp. 381–395, Springer (2011), https://doi.org/10.1007/978-3-642-19805-2_26

[17] Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1), 143–184 (1993), https://doi.org/10.1145/138027.138060

[18] Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. Journal of Functional Programming **17**(4–5), 613–673 (2007), https://doi.org/10.1017/S0956796807006430

[19] Haselwarter, P.G., Bauer, A.: Finitary type theories with and without contexts (2021), https://doi.org/10.48550/arXiv.2112.00539

[20] Leivant, D.: Typing and computational properties of lambda expressions. Theoretical Computer Science **44**, 51–68 (1986), https://doi.org/10.1016/0304-3975(86)90109-X

[21] McBride, C.: First-order unification by structural recursion. Journal of Functional Programming **13**(6), 1061–1075 (2003), https://doi.org/10.1017/S0956796803004957

[22] McBride, C.: First-order unification by structural recursion: correctness proof (2003), URL http://www.strictlypositive.org/foubsr-website/

[23] Moggi, E.: Computational lambda-calculus and monads. In: Symposium on Logic in Computer Science (LICS), pp. 14–23, IEEE (1989), https://doi.org/10.1109/LICS.1989.39155

[24] Pierce, B.C., Turner, D.N.: Local type inference. ACM Transactions on Programming Languages and Systems **22**(1), 1–44 (2000), https://doi.org/10.1145/345099.345100

[25] Reed, J.: Redundancy elimination for LF. In: International Workshop on Logical Frameworks and Meta-Languages (LFM 2004), vol. 199, pp. 89–106 (2008), https://doi.org/10.1016/j.entcs.2007.11.014

[26] Tanaka, M., Power, A.J.: Pseudo-distributive laws and axiomatics for variable binding. Higher-Order and Symbolic Computation **19**(2–3), 305–337 (2006), https://doi.org/10.1007/s10990-006-8750-x

[27] Tanaka, M., Power, J.: A unified category-theoretic semantics for binding signatures in substructural logic. Journal of Logic and Computation **16**(1), 5–25 (2006), https://doi.org/10.1093/logcom/exi070

[28] Uemura, T.: Abstract and Concrete Type Theories. Ph.D. thesis, University of Amsterdam (2021), URL https://hdl.handle.net/11245.1/41ff0b60-64d4-4003-8182-c244a9afab3b

[29] Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. Institute for Advanced Study (2013), URL https://homotopytypetheory.org/book

[30] Wadler, P., Kokke, W., Siek, J.G.: Programming Language Foundations in Agda (2022), URL https://plfa.inf.ed.ac.uk/22.08/

[31] Wells, J.B.: Typability and type checking in System F are equivalent and undecidable. Annals of Pure and Applied Logic **98**(1–3), 111–156 (1999), https://doi.org/10.1016/S0168-0072(98)00047-5

[32] Wood, J., Atkey, R.: A framework for substructural type systems. In: European Symposium on Programming (ESOP), Lecture Notes in Computer Science, vol. 13240, pp. 376–402, Springer (2022), https://doi.org/10.1007/978-3-030-99336-8_14

Generic bidirectional typing for dependent type theories

Thiago Felicissimo(✉)

Université Paris-Saclay, INRIA, LMF, ENS Paris-Saclay, Gif-sur-Yvette, France
thiago.felicissimo@inria.fr

Abstract. Bidirectional typing is a discipline in which the typing judgment is decomposed explicitly into inference and checking modes, allowing to control the flow of type information in typing rules and to specify algorithmically how they should be used. Bidirectional typing has been fruitfully studied and bidirectional systems have been developed for many type theories. However, the formal development of bidirectional typing has until now been kept confined to specific theories, with general guidelines remaining informal. In this work, we give a generic account of bidirectional typing for a general class of dependent type theories. This is done by first giving a general definition of type theories (or equivalently, a logical framework), for which we define declarative and bidirectional type systems. We then show, in a theory-independent fashion, that the two systems are equivalent. This equivalence is then explored to establish the decidability of typing for weak normalizing theories, yielding a generic type-checking algorithm that has been implemented in a prototype and used in practice with many theories.

Keywords: Type Theory · Bidirectional Typing · Logical Frameworks

1 Introduction

Algebraic [13,7] and logical framework [27,45,8,21] presentations of dependent type theories suffer from the verbosity of the required explicit type annotations, which destroys any hope of practical usability. In these settings, every type argument must be explicitly spelled out: an application is written as $t@_{A,x.B}u$, a dependent pair as $\langle t, u\rangle_{A,x.B}$, cons as $t ::_A l$ and the list goes on.

In order to restore usability, standard presentations of dependent type theories omit the majority of these annotations, so one writes $t\ u$ for application, $\langle t, u\rangle$ for a dependent pair, $t :: l$ for cons, etc. This unannotated syntax is so common that readers not familiar with algebraic presentations of type theory might not even realize that an omission is being made.

The omission of type arguments has nevertheless a cost: because knowing them is still important when typing terms, it becomes unclear how to do this algorithmically, even when decidability of conversion holds. Take for instance the typing rule for the dependent pair: in order to type $\langle t, u\rangle$ one needs to guess

S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 143–170, 2024.
https://doi.org/10.1007/978-3-031-57262-3_6

the arguments A and B, as unlike for the fully-annotated version $\langle t, u\rangle_{A,x.B}$ these are not stored in the syntax.

$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma, x : A \vdash B \ \mathsf{type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t, u\rangle : \Sigma x : A.B}$$

A solution to this problem is provided by *bidirectional typing* [35,16,19,20,41]. In this typing discipline, the declarative typing judgment $\Gamma \vdash t : A$ is decomposed explicitly into inference $\Gamma \vdash t \Rightarrow A$, where Γ and t are inputs and A is an output, and checking $\Gamma \vdash t \Leftarrow A$, where Γ, t and A are all inputs. The important point is that, by using these new judgments to control the flow of type information in a typing rule, one can specify algorithmically how the rule should be used. For instance, the following rule clarifies how one should type $\langle t, u\rangle$: the types A and B are not to be guessed, but instead recovered from the type C, which should be given as input.

$$\frac{C \longrightarrow^{*} \Sigma x : A.B \qquad \Gamma \vdash t \Leftarrow A \qquad \Gamma \vdash u \Leftarrow B[t/x]}{\Gamma \vdash \langle t, u\rangle \Leftarrow C}$$

We therefore see that bidirectional typing is the natural companion for an unannotated syntax, as it allows to algorithmically explain how the missing information can be retrieved.

Bidirectional typing has been fruitfully studied and bidirectional systems have been developed for many type theories [16,33,25,1,38,2]. However, the formal development of bidirectional typing has until now remained confined to specific theories, with general guidelines remaining informal. One can then naturally wonder if it would be possible to define a framework in which bidirectional typing could be studied generically, putting its general theory in solid ground. This is exactly the goal of this paper.

We contribute a theory-independent account of bidirectional typing. For this, we start by giving a general definition of type theories (or equivalently, a logical framework), which differs from previous frameworks [45,29] by allowing for the usual unannotated syntaxes most often used in practice. Each such theory then defines a declarative type system, which is shown to satisfy good properties like weakening, substitution and, for well-behaved theories, subject reduction.

Then, to formulate our bidirectional system we first address the known problem that some unannotated terms cannot be algorithmically typed [18]. We propose generic notions of *inferable* and *checkable* terms which, for non-degenerate theories, coincide respectively with neutrals and normal forms, and then formulate our bidirectional system for this subset of terms. As argued in previous works [38,1], this restriction is reasonable because users of type theory usually only write terms in normal form. We then show, in a theory-independent fashion, that the bidirectional system is sound and complete with respect to the declarative system, establishing an equivalence between the two.

This equivalence is then explored to establish, for weak normalizing theories, the decidability of inference for inferable terms, and the decidability of checking for checkable terms. This proof gives rise to generic type inference and checking algorithms, which have been implemented in a prototype theory-independent checker, allowing them to be used in practice with multiple theories. This implementation is described in detail in an accompanying experience report [22].

Plan of the paper We start in Section 2 by giving our general definition of type theories. We then move to Section 3, in which we give the declarative type system and show it satisfies nice properties. Section 4 then gives the bidirectional type system and its proof of equivalence with the declarative one. In Section 5 we show various examples of theories which are instances of our framework. We finish by discussing related work in Section 6 and then we conclude with Section 7.

2 General type theories

In this section, we give a general definition of type theories (or equivalently, a logical framework) for which we will give declarative and bidirectional type systems in later sections. Our definition is inspired by recent proposals of general definitions of type theories [29,45] and by the logical framework literature [27,8,26,40]. Our definition however crucially departs from these works by allowing for unannotated syntaxes and by exploring the constructor/destructor symmetries of symbols and rules in type theories.

We start the section by defining the raw syntax of our theories. Then, after defining patterns and substitution, we give one of the central definitions of our work: the one of theory. We finish the section by describing the rewrite judgment used to specify the definitional equality of our type theories.

2.1 Raw Syntax

Scopes and signatures The basic ingredients of our raw expressions are variables, metavariables and symbols. These are specified by scopes and signatures, which we define by the following grammars.

$$\boxed{\mathsf{Scope}} \ni \qquad \gamma, \delta ::= \cdot \mid \gamma, x$$

$$\boxed{\mathsf{MScope}} \ni \qquad \theta, \xi ::= \cdot \mid \theta, \mathsf{x}\{\delta\}$$

$$\boxed{\mathsf{Sig}} \ni \qquad \Sigma ::= \cdot \mid \Sigma, c(\theta) \mid \Sigma, d(\theta)$$

Let us go through the definition. A *(variable) scope* γ is simply a list of variables, whereas a *metavariable scope* θ is a list of metavariables, each being accompanied by a variable scope δ explaining the arguments each metavariable expects. A *signature* Σ then assigns a metavariable scope θ to each symbol, also explaining the arguments it expects.

We see that, from the start, we split symbols in two classes: *constructors* c and *destructors* d. This separation will be justified by the fact that each of these two classes will play a different role in our theories: destructors are the source of computation and are bidirectionally typed in mode infer, whereas constructors are the results of computations and are bidirectionally typed in mode check.

In the following, we write $\gamma.\delta$ and $\theta.\xi$ for scope concatenation, and we write constructor names in blue and destructor names in orange.

Example 1. The following signature $\Sigma_{\lambda\Pi}$ defines the raw syntax of a minimalistic Martin-Löf Type Theory (MLTT) with only dependent functions.

$$\mathrm{Ty},\ \mathrm{Tm}(\mathsf{A}),\ \Pi(\mathsf{A},\mathsf{B}\{x\}),\ \lambda(\mathsf{t}\{x\}),\ @(\mathsf{u}) \qquad (\Sigma_{\lambda\Pi})$$

The entry for @ might seem a bit strange since application usually takes two arguments, t and u. As we will see, destructors automatically take an extra argument, so we do not need to specify one for t. The reason for including symbols Ty and Tm will also become clear later.

Terms and spines Given a fixed signature Σ, we then define the terms, (variable) substitutions and metavariable substitutions by the following grammars.

$$\begin{array}{rrll}
\boxed{\mathsf{Tm}\ \theta\ \gamma} \ni & t,u,T,U ::= & \mid x & \text{if } x \in \gamma \\
& & \mid \mathsf{x}\{\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta\} & \text{if } \mathsf{x}\{\delta\} \in \theta \\
& & \mid c(\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi) & \text{if } c(\xi) \in \Sigma \\
& & \mid d(t \in \mathsf{Tm}\ \theta\ \gamma; \mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi) & \text{if } d(\xi) \in \Sigma \\
\boxed{\mathsf{Sub}\ \theta\ \gamma\ \delta} \ni & \vec{t},\vec{u},\vec{s},\vec{v} ::= & \mid \varepsilon & \text{if } \delta = \cdot \\
& & \mid \vec{u} \in \mathsf{Sub}\ \theta\ \gamma\ \delta', t \in \mathsf{Tm}\ \theta\ \gamma & \text{if } \delta = \delta', x \\
\boxed{\mathsf{MSub}\ \theta\ \gamma\ \xi} \ni & \mathbf{t},\mathbf{u},\mathbf{s},\mathbf{v} ::= & \mid \varepsilon & \text{if } \xi = \cdot \\
& & \mid \mathbf{u} \in \mathsf{MSub}\ \theta\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{Tm}\ \theta\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\}
\end{array}$$

We go through the definition step by step. First, we elect an intrinsically-scoped presentation of syntax, so the definitions of terms, substitutions and metavariable substitutions are each indexed by a scope γ and a metavariable scope θ, describing the variables and metavariables that can appear free.

A *term* is then either a variable x, a metavariable x applied to a substitution $\vec{t}$, a constructor c applied to a metavariable substitution $\mathbf{t}$ or a destructor d applied to a term t and a metavariable substitution $\mathbf{t}$. At the level of the syntax, the main difference between constructors and destructors is that the latter are automatically applied to a first argument, called the *principal argument*. Note also that we require each variable and metavariable to be in scope, and each metavariable or symbol to be applied to a substitution matching its scope of arguments, as specified by θ or Σ.

A *(variable) substitution* $\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ is then either the empty substitution when δ is empty, or a substitution $\vec{u} \in \mathsf{Sub}\ \theta\ \gamma\ \delta'$ and a term $t \in \mathsf{Tm}\ \theta\ \gamma$ when $\delta = \delta', x$. Therefore, we see that the scope δ describes the output (or the domain of definition) of the substitution $\vec{t}$. Similarly, a *metavariable substitution* $\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ is either empty when ξ is empty, or a metavariable substitution

$\mathbf{u} \in \mathsf{MSub}\ \theta\ \gamma\ \xi'$ and a term $t \in \mathsf{Tm}\ \theta\ \gamma.\delta$ when $\xi = \xi', \mathsf{x}\{\delta\}$. Unlike variable substitutions, metavariable substitutions are allowed to extend the scope of their arguments by binding the variables in δ, which we refer to by $\vec{x}_\delta$. This is used for instance in the cases of λ and Π in the following example.

Example 2. The terms defined by the signature $\Sigma_{\lambda\Pi}$ are given by the following grammar, where we omit the scope requirements for variables and metavariables.

$$t, u, A, B ::= x \mid \mathsf{x}\{\vec{t}\} \mid \mathrm{Ty} \mid \mathrm{Tm}(A) \mid \lambda(x.t) \mid \Pi(A, x.B) \mid @(t; u)$$

In the following, given a metavariable substitution $\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ and $\mathsf{x}\{\delta\} \in \xi$, we write $\mathbf{t}_\mathsf{x} \in \mathsf{Tm}\ \theta\ \gamma.\delta$ for the term in $\mathbf{t}$ at the position pointed by x. Similarly, given a substitution $\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ and $x \in \delta$, we write $t_x \in \mathsf{Tm}\ \theta\ \gamma$ for the term in $\vec{t}$ at the position pointed by x.

Contexts Given a fixed signature Σ, we define (variable) contexts and metavariable contexts by the following grammars.

$$\boxed{\mathsf{Ctx}\ \theta\ \gamma} \ni \quad \Gamma, \Delta ::= \cdot \mid \Gamma \in \mathsf{Ctx}\ \theta\ \gamma, x : T \in \mathsf{Tm}\ \theta\ \gamma.|\Gamma|$$

$$\boxed{\mathsf{MCtx}\ \theta} \ni \quad \Theta, \Xi ::= \cdot \mid \Theta \in \mathsf{MCtx}\ \theta, \mathsf{x}\{\Gamma \in \mathsf{Ctx}\ \theta.|\Theta|\ (\cdot)\} : T \in \mathsf{Tm}\ \theta.|\Theta|\ |\Gamma|$$

These are specified mutually with the *underlying scopes* $|\Gamma|$ of Γ and $|\Theta|$ of Θ, defined by the following clauses.

$$\begin{array}{ll} |\_| : \mathsf{Ctx}\ \theta\ \gamma \to \mathsf{Scope} & |\_| : \mathsf{MCtx}\ \theta \to \mathsf{MScope} \\ |\cdot| := \cdot & |\cdot| := \cdot \\ |\Gamma, x : T| := |\Gamma|, x & |\Theta, \mathsf{x}\{\Delta\} : T| := |\Theta|, \mathsf{x}\{|\Delta|\} \end{array}$$

A *context* $\Gamma \in \mathsf{Ctx}\ \gamma\ \theta$ is either empty, or composed by a context $\Gamma' \in \mathsf{Ctx}\ \gamma\ \theta$ and a variable x with a term $T \in \mathsf{Tm}\ \theta\ \gamma.|\Gamma'|$. The first important thing to note is that the term T does not live in scope γ, but in the extension of γ with the underlying scope of Γ', meaning that each entry in the context has access to the previously declared variables. Second, like in other frameworks [45], terms can also play the role of judgments, as illustrated by the following example.

Example 3. In MLTT we have two judgment forms: $\square\ \mathsf{type}$ for classifying types, and $\square : A$ for classifying terms. In our framework, these are represented by the constructors Ty and Tm. For instance, the context $A\ \mathsf{type}, x : A, y : (\Pi z : A.A)$ of MLTT is represented in our framework as

$$A : \mathrm{Ty}, x : \mathrm{Tm}(A), y : \mathrm{Tm}(\Pi(A, z.A))$$

which is syntactically well-formed in the signature $\Sigma_{\lambda\Pi}$. We can also write some strange contexts like $x : \lambda(z.z), y : x$, which will be eliminated later by typing.

The case of a *metavariable context* $\Theta \in \mathsf{MCtx}\ \theta$ is similar. We have either Θ empty or $\Theta = \Theta', \mathsf{x}\{\Delta\} : T$, where Δ has access to metavariables in θ and Θ, and T has moreover access to the variables in Δ.

Notation 1. We finish this subsection by establishing some notations.

- We write $e \in \mathsf{Expr}\ \theta\ \gamma$ as an informal abbreviation for any of the following: $e \in \mathsf{Tm}\ \theta\ \gamma$ or $e \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ or $e \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ or $e \in \mathsf{Ctx}\ \theta\ \gamma$.
- If the underlying signature is not clear from the context, we write Tm_Σ, Sub_Σ, MSub_Σ, Ctx_Σ, MCtx_Σ in order to make it explicit.
- We write $\mathsf{Ctx}\ \theta$ for $\mathsf{Ctx}\ \theta\ (\cdot)$, Ctx for $\mathsf{Ctx}\ (\cdot)\ (\cdot)$ and MCtx for $\mathsf{MCtx}\ (\cdot)$.

Remark 1. Because we work with a nameful syntax, we allow ourselves to implicitly weaken expressions: if $e \in \mathsf{Expr}\ \theta\ \gamma$ and θ is a subsequence of θ' and γ is a subsequence of γ' then we also have $e \in \mathsf{Expr}\ \theta'\ \gamma'$. Nevertheless, we expect that our proofs can be formally carried out using de Bruijn indices, by properly inserting weakenings whenever needed, and showing all the associated lemmata.

2.2 Substitution

Before defining the application of a substitution to a term, we first need to define the *identity substitutions*, by the following clauses. Note that, while the identity variable substitution id_γ is just the list of variables from γ, the identity metavariable substitution id_θ needs to eta-expand each metavariable $\mathsf{x}\{\delta\} \in \theta$ to $\vec{x}_\delta.\mathsf{x}\{\mathsf{id}_\delta\}$ in order for the result to be a valid metavariable substitution. In the following, we sometimes abuse notation and write id_Γ for $\mathsf{id}_{|\Gamma|}$ and id_Θ for $\mathsf{id}_{|\Theta|}$.

$$\begin{array}{ll}
\mathsf{id}\_ : (\gamma \in \mathsf{Scope}) \to \mathsf{Sub}\ (\cdot)\ \gamma\ \gamma & \mathsf{id}\_ : (\theta \in \mathsf{MScope}) \to \mathsf{MSub}\ \theta\ (\cdot)\ \theta \\
\mathsf{id}_{(\cdot)} := \varepsilon & \mathsf{id}_{(\cdot)} := \varepsilon \\
\mathsf{id}_{\gamma,x} := \mathsf{id}_\gamma, x & \mathsf{id}_{\theta,\mathsf{x}\{\gamma\}} := \mathsf{id}_\theta, \vec{x}_\gamma.\mathsf{x}\{\mathsf{id}_\gamma\}
\end{array}$$

We can now define in Figure 1 the application of a substitution to an expression. Given a variable substitution $\vec{v} \in \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2$ its application to an expression $e \in \mathsf{Expr}\ \theta\ \gamma_2$ gives $e[\vec{v}] \in \mathsf{Expr}\ \theta\ \gamma_1$, and given a metavariable substitution $\mathbf{v} \in \mathsf{MSub}\ \theta_1\ \delta\ \theta_2$ its application to an expression $e \in \mathsf{Expr}\ \theta_2\ \gamma$ gives $e[\mathbf{v}] \in \mathsf{Expr}\ \theta_1\ \delta.\gamma$. The main case of the definition is when we substitute $\mathbf{v} \in \mathsf{MSub}\ \theta_1\ \delta\ \theta_2$ in the term $\mathsf{x}\{\vec{t}\} \in \mathsf{Tm}\ \theta_2\ \gamma$. If $\mathsf{x}\{\gamma_\mathsf{x}\} \in \theta_2$, then by recursively substituting $\mathbf{v}$ in $\vec{t} \in \mathsf{Sub}\ \theta_2\ \gamma\ \gamma_\mathsf{x}$ we get $\vec{t}[\mathbf{v}] \in \mathsf{Sub}\ \theta_1\ \delta.\gamma\ \gamma_\mathsf{x}$. We moreover have $\mathbf{v}_\mathsf{x} \in \mathsf{Tm}\ \theta_1\ \delta.\gamma_\mathsf{x}$, so by substituting the variables in γ_x by $\vec{t}[\mathbf{v}]$ and the ones in δ by id_δ we get $\mathbf{v}_\mathsf{x}[\mathsf{id}_\delta, \vec{t}[\mathbf{v}]] \in \mathsf{Tm}\ \theta_1\ \delta.\gamma$ as the final result.

Example 4. If $t \in \mathsf{Tm}\ (\cdot)\ (\gamma, x)$ and $u \in \mathsf{Tm}\ (\cdot)\ \gamma$, then by applying $x.t, u \in \mathsf{MSub}\ (\cdot)\ \gamma\ (\mathsf{t}\{x\}, \mathsf{u})$ to $@(\lambda(x.\mathsf{t}\{x\}); \mathsf{u}) \in \mathsf{Tm}\ (\mathsf{t}\{x\}, \mathsf{u})\ (\cdot)$ we get the term $@(\lambda(x.t); u) \in \mathsf{Tm}\ (\cdot)\ \gamma$. Note in particular that, compared to frameworks derived from contextual modal type theory [37], our metavariable substitutions are not required to be closed and can introduce new variables in the scope of the resulting term. For instance, while the inital term $@(\lambda(x.\mathsf{t}\{x\}); \mathsf{u})$ lives in an empty variable scope, the resulting term $@(\lambda(x.t); u)$ lives in the variable scope γ.

Substitution application satisfies all the expected laws, such as $e[\mathbf{v}][\mathbf{u}] = e[\mathbf{v}[\mathbf{u}]]$, $e[\mathsf{id}_\gamma] = e$ and $\mathsf{id}_\theta[\mathbf{v}] = \mathbf{v}$ — we refer to the technical report [24] for a detailed account of these properties, and for the full definition of substitution.

$$
\begin{array}{ll}
\_[\_] : \mathsf{Tm}\ \theta\ \gamma_2 \to \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2 & \_[\_] : \mathsf{Tm}\ \theta_2\ \gamma \to \mathsf{MSub}\ \theta_1\ \delta\ \theta_2 \\
\quad \to \mathsf{Tm}\ \theta\ \gamma_1 & \quad \to \mathsf{Tm}\ \theta_1\ \delta.\gamma \\
x[\vec{v}] := v_x & x[\mathbf{v}] := x \\
\mathsf{x}\{\vec{t}\}[\vec{v}] := \mathsf{x}\{\vec{t}[\vec{v}]\} & \mathsf{x}\{\vec{t}\}[\mathbf{v}] := \mathbf{v}_{\mathsf{x}}[\mathsf{id}_\delta, \vec{t}[\mathbf{v}]] \\
c(\mathbf{t})[\vec{v}] := c(\mathbf{t}[\vec{v}]) & c(\mathbf{t})[\mathbf{v}] := c(\mathbf{t}[\mathbf{v}]) \\
d(t;\mathbf{u})[\vec{v}] := d(t[\vec{v}];\mathbf{u}[\vec{v}]) & d(t;\mathbf{u})[\mathbf{v}] := d(t[\mathbf{v}];\mathbf{u}[\mathbf{v}])
\end{array}
$$

Fig. 1. Application of a variable or metavariable substitution (excerpt)

2.3 Patterns

There will be a need to isolate a special class of expressions that will be shown later to support decidable and unitary matching. This will be needed both to define the rewrite rules of our theories and to determine when omitted arguments can be recovered. For this, given a fixed signature Σ, we define the *term patterns* and *metavariable substitution patterns* by the following grammars.

$$
\begin{array}{lll}
\boxed{\mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma} \ni t, u ::= & \mid \mathsf{x}\{\mathsf{id}_\gamma\} & \text{if } \theta = \mathsf{x}\{\gamma\} \\
& \mid c(\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi) & \text{if } c(\xi) \in \Sigma \\
\boxed{\mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi} \ni \mathbf{t}, \mathbf{u} ::= & \mid \varepsilon & \text{if } \xi = \cdot \text{ and } \theta = \cdot \\
& \mid \mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta_1\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{Tm}^{\mathsf{P}}\ \theta_2\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\} \\
& & \quad \text{and } \theta = \theta_1.\theta_2
\end{array}
$$

As we can see, the only symbols that are allowed to appear in patterns are constructors. This will be essential later to ensure that patterns do not only support syntactic matching, but also matching modulo rewriting. Moreover, our patterns are linear, and so each metavariable in scope occurs exactly once. Finally, our patterns are *fully-applied*, meaning that each metavariable occurrence is fully applied to all variables in scope.

Example 5. In the signature $\Sigma_{\lambda\Pi}$ we can build the pattern

$$\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \in \mathsf{Tm}^{\mathsf{P}}\ (\mathsf{A},\ \mathsf{B}\{x\})\ (\cdot)$$

We have the inclusions $\mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma \subset \mathsf{Tm}\ \theta\ \gamma$ and $\mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi \subset \mathsf{MSub}\ \theta\ \gamma\ \xi$, which we use to implicitly coerce patterns into regular expressions when needed.

2.4 Theories

We now come to a central definition in our work, that of a *theory* $\mathbb{T}$. We define inductively how a theory is built, simultaneously with its underlying signature $|\mathbb{T}|$ — technically, our definition is by small induction-recursion. The base case covers the empty theory $\mathbb{T} = (\cdot)$, so assuming now a theory $\mathbb{T}$ is given we can extend it with two types of entries: *schematic typing rules* and *rewrite rules*. We start with the first, which come in three kinds: sort, constructor and destructor rules.

Sort rules In our framework, a *sort* T is a term that can appear to the right of a colon.[1] As hinted in Example 3, and following [13,45,44], sorts are used to specify the judgment forms of a theory. For instance, the two judgment forms of MLTT "$\square$ type" and "$\square : A$" are defined by the following.

$$\frac{}{\vdash \mathrm{Ty}\ \mathsf{sort}} \qquad \frac{\vdash \mathsf{A} : \mathrm{Ty}}{\vdash \mathrm{Tm}(\mathsf{A})\ \mathsf{sort}}$$

Formally, a sort rule is of the form

$$c(\Xi \in \mathsf{MCtx}_{|\mathbb{T}|})\ \mathsf{sort}$$

and the previously shown rules are just an informal notation for $\mathrm{Ty}(\cdot)$ sort and $\mathrm{Tm}(\mathsf{A} : \mathrm{Ty})$ sort. In the following, we will make use of such informal representations in order to enhance readability of schematic rules. Note also that the premises (e.g. $\vdash \mathsf{A} : \mathrm{Ty}$) correspond to entries in the metavariable context of the rule, and henceforth we will use these two points of view interchangeably.

Constructor rules Like most works in bidirectional typing [35,19,1,16], our framework imposes that constructors are to be bidirectionally typed in mode check, and thus the sort of the conclusion can be used to recover arguments which are not recorded in the syntax. To capture this, premises are split into two metavariable contexts Ξ_1 and Ξ_2, where Ξ_1 is erased and Ξ_2 is stored in the term. The sort T is then required to be a pattern containing the metavariables of Ξ_1, leading to constructor rules of the following form.

$$c(\Xi_1 \in \mathsf{MCtx}_{|\mathbb{T}|};\ \Xi_2 \in \mathsf{MCtx}_{|\mathbb{T}|}\ |\Xi_1|) : U \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ |\Xi_1|\ (\cdot)$$

Two examples of constructor rules are the ones for Π and λ — note however that the one for Π is slightly degenerate, given that we have $\Xi_1 = \cdot$ and thus no erased premises.

$$\frac{\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty}}{\vdash \Pi(\mathsf{A}, \mathsf{B}) : \mathrm{Ty}} \qquad \frac{\begin{array}{c}\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})\end{array}}{\vdash \lambda(\mathsf{t}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))}$$

These are just informal notations for $\Pi(\cdot;\ \mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}) : \mathrm{Ty}$ and $\lambda(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$.

Destructor rules As for constructor rules, in destructor rules premises are also separated into erased Ξ_1 and non-erased Ξ_2 parts. However, unlike constructors, destructors are bidirectionally typed in inference mode. In this case, the erased arguments in Ξ_1 are not to be recovered from the sort of the conclusion, but instead by inferring the sort of the *principal argument* which is required to be

[1] We avoid calling them "types" to prevent a name clash with the types of the theories we define. Still, we allow ourselves to say "t is typed by sort T" to mean $t : T$.

a pattern containing the metavariables of Ξ_1. The destructor rules are therefore of the following form.

$$d(\Xi_1 \in \mathsf{MCtx}_{|\mathbb{T}|};\ \mathsf{x} : T \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ |\Xi_1|\ (\cdot);\ \Xi_2 \in \mathsf{MCtx}_{|\mathbb{T}|}\ (|\Xi_1|, \mathsf{x}))$$
$$: U \in \mathsf{Tm}_{|\mathbb{T}|}\ (|\Xi_1|, \mathsf{x}, |\Xi_2|)\ (\cdot)$$

The main example of destructor rule is the application rule

$$\frac{\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \vdash \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \vdash \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{\vdash @(\mathsf{t}; \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})}$$

which is just an informal notation for

$$@(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$$

Rewrite rules Finally, the last type of rules are rewrite rules, which allow us to specify the definitional equality (also known as conversion) of the theory. As suggested by our constructor/destructor separation of symbols, the left-hand side of rewrite rules are to be headed by destructors. Moreover, to ensure the decidability of rewriting, we also ask both arguments t and $\mathbf{u}$ of the left hand side $d(t; \mathbf{u})$ to be patterns. The right-hand side of the rule is then a term containing only the metavariables introduced by t and $\mathbf{u}$. The rewrite rules are hence of the following form, where $d(\xi) \in |\mathbb{T}|$.

$$\theta_1; \theta_2 \Vdash d(t \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ \theta_1\ (\cdot); \mathbf{u} \in \mathsf{MSub}^{\mathsf{P}}_{|\mathbb{T}|}\ \theta_2\ (\cdot)\ \xi) \longmapsto r \in \mathsf{Tm}_{|\mathbb{T}|}\ \theta_1.\theta_2\ (\cdot)$$

We shall also ask for a supplementary condition: in order to extend a theory $\mathbb{T}$ with a rule $\theta_1; \theta_2 \Vdash l \longmapsto r$, there can be no rule $\theta_1'; \theta_2' \Vdash l' \longmapsto r'$ in $\mathbb{T}$ such that we have $l[\mathbf{v}] = l'[\mathbf{v}']$ for some $\mathbf{v}$ and $\mathbf{v}'$. As discussed in the next subsection, this will ensure that the rewrite system is confluent by construction.

The prototypical example of a rewrite rule is the computation rule for functions: the β-rule from the λ-calculus.

$$\mathsf{t}\{x\}; \mathsf{u} \Vdash @(\lambda(x.\mathsf{t}\{x\}); \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$$

In the following we allow ourselves to omit the metavariable scopes $\theta_1; \theta_2$ as these can be easily reconstructed by inspecting the rewrite rule's left hand side.

Underlying signature Finally, the recursive definition of the underlying signature of a theory is given by the following clauses, where we write Thy for the set of theories. As we can see, in both constructor and destructor rules the metavariable context of erased premises Ξ_1 is omitted from the syntax. Moreover, rewrite rules are simply ignored when computing the underlying signature.

$$\begin{array}{ll}
|\_| : \mathsf{Thy} \to \mathsf{Sig} & |\mathbb{T}, c(\Xi_1; \Xi_2) : U| := |\mathbb{T}|, c(|\Xi_2|) \\
|\cdot| := \cdot & |\mathbb{T}, d(\Xi_1; \mathsf{x} : T; \Xi_2) : U| := |\mathbb{T}|, d(|\Xi_2|) \\
|\mathbb{T}, c(\Xi)\ \mathsf{sort}| := |\mathbb{T}|, c(|\Xi|) & |\mathbb{T}, \theta_1; \theta_2 \Vdash d(t; \mathbf{u}) \longmapsto r| := |\mathbb{T}|
\end{array}$$

Example 6. By putting together all of the rules previously seen in this subsection, we get the following theory $\mathbb{T}_{\lambda\Pi}$ defining a basic version of MLTT with only dependent functions.

$$\begin{aligned}&\mathrm{Ty}(\cdot)\ \mathsf{sort},\quad \mathrm{Tm}(\mathsf{A}:\mathrm{Ty})\ \mathsf{sort},\quad \Pi(\cdot;\ \ \mathsf{A}:\mathrm{Ty},\ \mathsf{B}\{x:\mathrm{Tm}(\mathsf{A})\}:\mathrm{Ty}):\mathrm{Ty},\\ &\lambda(\mathsf{A}:\mathrm{Ty},\ \mathsf{B}\{x:\mathrm{Tm}(\mathsf{A})\}:\mathrm{Ty};\ \ \mathsf{t}\{x:\mathrm{Tm}(\mathsf{A})\}:\mathrm{Tm}(\mathsf{B}\{x\})):\mathrm{Tm}(\Pi(\mathsf{A},x.\mathsf{B}\{x\})),\\ &@(\mathsf{A}:\mathrm{Ty},\ \mathsf{B}\{x:\mathrm{Tm}(\mathsf{A})\}:\mathrm{Ty};\ \ \mathsf{t}:\mathrm{Tm}(\Pi(\mathsf{A},x.\mathsf{B}\{x\}));\ \ \mathsf{u}:\mathrm{Tm}(\mathsf{A})):\mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}),\\ &@(\lambda(x.\mathsf{t}\{x\});\mathsf{u})\longmapsto \mathsf{t}\{\mathsf{u}\}\end{aligned}\qquad (\mathbb{T}_{\lambda\Pi})$$

When computing its underlying signature $|\mathbb{T}_{\lambda\Pi}|$ we get the signature $\Sigma_{\lambda\Pi}$.

2.5 Rewriting

The rewrite rules of a theory $\mathbb{T}$ are used to define the *rewrite relation* $e \longrightarrow e'$ for expressions $e, e' \in \mathsf{Expr}\ \theta\ \gamma$, which is given by the context closure of the following rule — see the technical report [24] for the full definition.

$$\frac{(\theta_1;\theta_2 \Vdash d(t;\mathbf{u}) \longmapsto r) \in \mathbb{T} \qquad \mathbf{v}_1 \in \mathsf{MSub}\ \theta\ \gamma\ \theta_1 \qquad \mathbf{v}_2 \in \mathsf{MSub}\ \theta\ \gamma\ \theta_2}{d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) \longrightarrow r[\mathbf{v}_1,\mathbf{v}_2] \in \mathsf{Tm}\ \theta\ \gamma}$$

The relations $\longrightarrow^{+}$, $\longrightarrow^{*}$ and $\equiv$ are then defined as usual, respectively as the transitive, reflexive-transitive and reflexive-symmetric-transitive closures of $\longrightarrow$. The relation $\equiv$ is called the *definitional equality* (or *conversion*) of the theory.

Rewriting satisfies the key property of being stable under substitution: if $e \longrightarrow^{*} e'$ and $\vec{v} \longrightarrow^{*} \vec{v}'$ then $e[\vec{v}] \longrightarrow^{*} e'[\vec{v}']$, and if $e \longrightarrow^{*} e'$ and $\mathbf{v} \longrightarrow^{*} \mathbf{v}'$ then $e[\mathbf{v}] \longrightarrow^{*} e'[\mathbf{v}']$. This implies in particular that definitional equality is also stable under substitution.

Finally, recall that when defining theories we asked that no two different left-hand sides should unify. Because this is the only way two rule can overlap, this means that there are no *critical pairs* [11]. Therefore, because rules are also all *left-linear*, it follows that the rewrite system of any theory is *orthogonal*, hence confluent by [34, Theorem 6.11].

Proposition 1 (Confluence). *If $e' \ {}^{*}\!\!\longleftarrow e \longrightarrow^{*} e''$ then there is some e''' such that $e' \longrightarrow^{*} e''' \ {}^{*}\!\!\longleftarrow e''$. In particular, this implies that whenever $e \equiv e'$ then we have $e \longrightarrow^{*} e'' \ {}^{*}\!\!\longleftarrow e'$ for some e''.*

Notation 2. Whenever the underlying theory is not clear from the context we write $\longrightarrow_{\mathbb{T}}$ and $\longrightarrow^{+}_{\mathbb{T}}$ and $\longrightarrow^{*}_{\mathbb{T}}$ and $\equiv_{\mathbb{T}}$ to make it explicit.

3 Declarative type system

In the previous section we have given a general definition of type theories. As explained in the introduction, each theory also defines a declarative type system, which can be seen as the platonic type system, and a bidirectional type system, which is the one that can be algorithmically used in practice.

In this section we introduce the declarative type system. This system is then used to define the *valid theories*, a class of theories which are well-behaved. We then conclude the section by showing that the declarative system satisfies nice properties, and in particular satisfies subject reduction when the theory is valid.

3.1 Declarative typing rules

Given a fixed theory $\mathbb{T}$, the declarative type system is defined by the rules in Figure 2. The system is split in 6 judgments:

- $\Theta \vdash$: Well-formedness of metavariable context Θ.
- $\Theta; \Gamma \vdash$: Well-formedness of variable context Γ under metavariable context Θ.
- $\Theta; \Gamma \vdash T \ \mathsf{sort}$: Well-formedness of sort T under contexts $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash t : T$: Typing of a term t by T under context $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash \vec{t} : \Delta$: Typing of a variable substitution $\vec{t}$ by Δ under context $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash \mathbf{t} : \Xi$: Typing of a metavariable substitution $\mathbf{t}$ by Ξ under context $\Theta; \Gamma$.

The most important rules are the ones which instantiate schematic typing rules: Cons, Dest and Sort. For instance, in order to use Dest to type $d(t; \mathbf{u})$ a metavariable substitution $\mathbf{t}$ not stored in the syntax must be "guessed", and then we must show that $\mathbf{t}, t, \mathbf{u}$ is typed by $\Xi_1.(\mathsf{x} : T).\Xi_2$. The rules for typing a metavariable substitution can then be applied, which has the effect of unfolding the judgment $\mathbf{t}, t, \mathbf{u} : \Xi_1.(\mathsf{x} : T).\Xi_2$ into regular term typing judgments. At the end of this unfolding process, the resulting "big-step derivation" has basically the same shape as the schematic typing rule for d, and it can be understood as its instantiation. Let us look at a concrete example of this.

Example 7. Suppose we want to show that $@(t; u)$ is well-typed in the theory $\mathbb{T}_{\lambda\Pi}$. Because $@$ is a destructor symbol with schematic rule

$$@(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$$

by guessing some A and B we can start the derivation with rule Dest, giving

$$\frac{\Theta; \Gamma \vdash A, x.B, t, u : (\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty},\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})),\ \mathsf{u} : \mathrm{Tm}(\mathsf{A}))}{\Theta; \Gamma \vdash @(t; u) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})[A, x.B, t, u]}$$

If we note that $\mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})[A, x.B, t, u] = \mathrm{Tm}(B[\mathsf{id}_\Gamma, u])$, and we continue by applying the rules defining the judgment $\Theta; \Gamma \vdash \mathbf{t} : \Xi$, we get

$$\frac{\begin{array}{ccc} \Theta; \Gamma \vdash & \Theta; \Gamma \vdash A : \mathrm{Ty} & \Theta; \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \\ \multicolumn{3}{c}{\Theta; \Gamma \vdash t : \mathrm{Tm}(\Pi(A, x.B)) \qquad \Theta; \Gamma \vdash u : \mathrm{Tm}(A)} \end{array}}{\Theta; \Gamma \vdash @(t; u) : \mathrm{Tm}(B[\mathsf{id}_\Gamma, u])}$$

which can be understood as the instantiation of the schematic rule for $@$. This also corresponds to the usual application rule of MLTT, as the first 3 premises can be shown admissible by inversion of typing and the results of Subsection 3.3.

$$\boxed{\Theta \vdash \quad (\Theta \in \mathsf{MCtx})} \qquad \boxed{\Theta; \Gamma \vdash \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|)}$$

$$\textsc{EmptyMCtx}\ \frac{}{\cdot \vdash} \qquad \textsc{ExtMCtx}\ \frac{\Theta; \Gamma \vdash T\ \mathsf{sort}}{\Theta, \mathsf{x}\{\Gamma\} : T \vdash} \qquad \textsc{EmptyCtx}\ \frac{\Theta \vdash}{\Theta; \cdot \vdash} \qquad \textsc{ExtCtx}\ \frac{\Theta; \Gamma \vdash T\ \mathsf{sort}}{\Theta; \Gamma, x : T \vdash}$$

$$\boxed{\Theta; \Gamma \vdash T\ \mathsf{sort} \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ T \in \mathsf{Tm}\ |\Theta|\ |\Gamma|)}$$

$$\textsc{Sort}\quad c(\Xi)\ \mathsf{sort} \in \mathbb{T}\ \frac{\Theta; \Gamma \vdash \mathbf{t} : \Xi}{\Theta; \Gamma \vdash c(\mathbf{t})\ \mathsf{sort}}$$

$$\boxed{\Theta; \Gamma \vdash t : T \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ T \in \mathsf{Tm}\ |\Theta|\ |\Gamma|;\ t \in \mathsf{Tm}\ |\Theta|\ |\Gamma|)}$$

$$\textsc{Cons}\quad c(\Xi_1; \Xi_2) : T \in \mathbb{T}\ \frac{\Theta; \Gamma \vdash \mathbf{t}, \mathbf{u} : \Xi_1.\Xi_2}{\Theta; \Gamma \vdash c(\mathbf{u}) : T[\mathbf{t}]} \qquad \textsc{Var}\quad x : T \in \Gamma\ \frac{\Theta; \Gamma \vdash}{\Theta; \Gamma \vdash x : T}$$

$$\textsc{Dest}\quad d(\Xi_1; \mathsf{x} : T; \Xi_2) : U \in \mathbb{T}\ \frac{\Theta; \Gamma \vdash \mathbf{t}, t, \mathbf{u} : \Xi_1.(\mathsf{x} : T).\Xi_2}{\Theta; \Gamma \vdash d(t; \mathbf{u}) : U[\mathbf{t}, t, \mathbf{u}]} \qquad \textsc{MVar}\quad \mathsf{x}\{\Delta\} : T \in \Theta\ \frac{\Theta; \Gamma \vdash \vec{t} : \Delta}{\Theta; \Gamma \vdash \mathsf{x}\{\vec{t}\} : T[\vec{t}]}$$

$$\textsc{Conv}\quad T \equiv_{\mathbb{T}} U\ \frac{\Theta; \Gamma \vdash t : T \qquad \Theta; \Gamma \vdash U\ \mathsf{sort}}{\Theta; \Gamma \vdash t : U}$$

$$\boxed{\Theta; \Gamma \vdash \vec{t} : \Delta \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ \Delta \in \mathsf{Ctx}\ |\Theta|;\ \vec{t} \in \mathsf{Sub}\ |\Theta|\ |\Gamma|\ |\Delta|)}$$

$$\textsc{EmptySub}\ \frac{\Theta; \Gamma \vdash}{\Theta; \Gamma \vdash \varepsilon : (\cdot)} \qquad \textsc{ExtSub}\ \frac{\Theta; \Gamma \vdash \vec{t} : \Delta \qquad \Theta; \Gamma \vdash t : T[\vec{t}]}{\Theta; \Gamma \vdash \vec{t}, t : (\Delta, x : T)}$$

$$\boxed{\Theta; \Gamma \vdash \mathbf{t} : \Xi \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ \Xi \in \mathsf{MCtx};\ \mathbf{t} \in \mathsf{MSub}\ |\Theta|\ |\Gamma|\ |\Xi|)}$$

$$\textsc{EmptyMSub}\ \frac{\Theta; \Gamma \vdash}{\Theta; \Gamma \vdash \varepsilon : (\cdot)} \qquad \textsc{ExtMSub}\ \frac{\Theta; \Gamma \vdash \mathbf{t} : \Xi \qquad \Theta; \Gamma.\Delta[\mathbf{t}] \vdash t : T[\mathbf{t}]}{\Theta; \Gamma \vdash \mathbf{t}, \vec{x}_\Delta.t : (\Xi, \mathsf{x}\{\Delta\} : T)}$$

Fig. 2. Declarative typing rules

Notation 3. We finish this subsection by establishing some notations.

1. We write $\Theta; \Gamma \vdash \mathcal{J}$ for any of the following: $\Theta; \Gamma \vdash$ or $\Theta; \Gamma \vdash T$ sort or $\Theta; \Gamma \vdash t : T$ or $\Theta; \Gamma \vdash \vec{t} : \Delta$ or $\Theta; \Gamma \vdash \mathbf{t} : \Xi$.
2. We write $\mathbb{T} \triangleright \Theta; \Gamma \vdash \mathcal{J}$ when $\mathbb{T}$ is not clear from the context.
3. We write $\Theta \vdash \mathcal{J}$ for $\Theta; \cdot \vdash \mathcal{J}$ and $\Gamma \vdash \mathcal{J}$ for $\cdot; \Gamma \vdash \mathcal{J}$.

3.2 Valid theories

Our definition of theories given in Subsection 2.4 is a bit too permissive, and we would like to isolate a class of theories for which we can show nicer properties. These are the *valid theories*, defined by the following inference rules.

$$\frac{}{\cdot \ \mathsf{valid}} \qquad \frac{\mathbb{T}\ \mathsf{valid} \qquad \mathbb{T} \triangleright \Xi_1 \vdash T\ \mathsf{sort} \qquad \mathbb{T} \triangleright \Xi_1.\Xi_2 \vdash}{\mathbb{T}, c(\Xi_1; \Xi_2) : T\ \mathsf{valid}}$$

$$\frac{\mathbb{T}\ \mathsf{valid} \qquad \mathbb{T} \triangleright \Xi \vdash}{\mathbb{T}, c(\Xi)\ \mathsf{sort\ valid}} \qquad \frac{\mathbb{T}\ \mathsf{valid} \qquad \mathbb{T} \triangleright \Xi_1.(\mathsf{x} : U).\Xi_2 \vdash T\ \mathsf{sort}}{\mathbb{T}, d(\Xi_1; \mathsf{x} : U; \Xi_2) : T\ \mathsf{valid}}$$

$$\frac{\begin{array}{c}\mathbb{T}\ \mathsf{valid} \qquad d(\Xi_1; \mathsf{x} : U; \Xi_2) : V \in \mathbb{T} \qquad \text{for some } \Theta_1, \Theta_2 \text{ with } |\Theta_1| = \theta_1, |\Theta_2| = \theta_2 : \\ \mathbb{T} \triangleright \Xi_1.\Theta_1.\Theta_2 \vdash (\mathsf{id}_{\Xi_1}, t, \mathbf{u}) : \Xi_1.(\mathsf{x} : U).\Xi_2 \qquad \mathbb{T} \triangleright \Xi_1.\Theta_1.\Theta_2 \vdash r : V[\mathsf{id}_{\Xi_1}, t, \mathbf{u}]\end{array}}{\mathbb{T}, \theta_1; \theta_2 \Vdash d(t; \mathbf{u}) \longmapsto r\ \mathsf{valid}}$$

Intuitively, the definition of valid theories ensures that each time we extend a theory $\mathbb{T}$ with a schematic typing or rewrite rule, $\mathbb{T}$ can justify that the extension is well-behaved. For sort rules $c(\Xi)$ sort this means ensuring that the metavariable context Ξ is well-formed in $\mathbb{T}$, and for destructor rules $d(\Xi_1; \mathsf{x} : U; \Xi_2) : T$ this means ensuring that T is a well-formed sort in metavariable context $\Xi_1.(\mathsf{x} : U).\Xi_2$ and in the theory $\mathbb{T}$. The rule for a constructor $c(\Xi_1; \Xi_2) : T$ does not only ask $\Xi_1.\Xi_2$ to be a well-formed metavariable context, but also for the term T to be a well-formed sort for Ξ_1 — recall that T can only contain metavariables from Ξ_1.

The most complicated case is for a rewrite rule $\theta_1; \theta_2 \Vdash d(t; \mathbf{u}) \longmapsto r$, in which we must find metavariable contexts Θ_1, Θ_2 with $|\Theta_1| = \theta_1$ and $|\Theta_2| = \theta_2$ allowing to type $\mathsf{id}_{\Xi_1}, t, \mathbf{u}$ and r. This technical condition is essential to prove subject reduction of our valid theories (Theorem 1).

Example 8. It is tedious but uncomplicated to see that the theory $\mathbb{T}_{\lambda\Pi}$ is valid. The most interesting part of the proof is when we add the β-rule

$$\mathsf{t}\{x\}; \mathsf{u} \Vdash @(\lambda(x.\mathsf{t}\{x\}); \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$$

If we write $\mathbb{T}'$ for the part of $\mathbb{T}_{\lambda\Pi}$ preceding this declaration, and (for space reasons) we abbreviate $(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(A)\} : \mathrm{Ty})$ as Θ_{AB}, we have to show

$$\mathbb{T}' \triangleright \Theta_{\mathsf{AB}}.\Theta_1.\Theta_2 \vdash (\mathsf{A}, x.\mathsf{B}\{x\}, \lambda(x.\mathsf{t}\{x\}), \mathsf{u}) : \Theta_{\mathsf{AB}}.(\mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})), \mathsf{u} : \mathrm{Tm}(\mathsf{A}))$$

and

$$\mathbb{T}' \triangleright \Theta_{\mathsf{AB}}.\Theta_1.\Theta_2 \vdash \mathsf{t}\{\mathsf{u}\} : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$$

which can both be easily shown if we chose $\Theta_1 = \mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})$ and $\Theta_2 = \mathsf{u} : \mathrm{Tm}(\mathsf{A})$, which indeed satisfy $|\Theta_1| = \mathsf{t}\{x\}$ and $|\Theta_2| = \mathsf{u}$.

3.3 Basic metaproperties

We now show some basic metaproperties satisfied by the declarative type system. The assumption that the theory is valid is not necessary for all results, and will be stated explicitly when needed. We give proof sketches for some of the properties, and refer to the technical report [24] for the proofs.

Proposition 2 (Contexts are well-formed). *The following rules are admissible.*

$$\frac{\Theta;\Gamma\vdash\mathcal{J}}{\Theta;\Gamma\vdash}\qquad\qquad\frac{\Theta;\Gamma\vdash\mathcal{J}}{\Theta\vdash}$$

Proposition 3 (Weakening). *Let us write $\Gamma\sqsubseteq\Delta$ if Γ is a subsequence of Δ, and $\Theta\sqsubseteq\Xi$ if Θ is a subsequence of Ξ. The following rules are admissible.*

$$\Gamma\sqsubseteq\Delta\,\frac{\Theta;\Gamma\vdash\mathcal{J}\qquad\Theta;\Delta\vdash}{\Theta;\Delta\vdash\mathcal{J}}\qquad\qquad\Theta\sqsubseteq\Xi\,\frac{\Theta;\Gamma\vdash\mathcal{J}\qquad\Xi\vdash}{\Xi;\Gamma\vdash\mathcal{J}}$$

In order to state the substitution property, given $\Theta;\Delta\vdash\mathcal{J}$ we define the notations $(\Theta;\Delta\vdash\mathcal{J})[\vec{v}]$ and $(\Theta;\Delta\vdash\mathcal{J})[\mathbf{v}]$ by the following table.

$\Theta;\Delta\vdash\mathcal{J}$	$(\Theta;\Delta\vdash\mathcal{J})[\vec{v}]$ where $\Theta;\Gamma\vdash\vec{v}:\Delta$	$(\Theta;\Delta\vdash\mathcal{J})[\mathbf{v}]$ where $\Xi;\Gamma\vdash\mathbf{v}:\Theta$
$\Theta;\Delta\vdash$	$\Theta;\Gamma\vdash$	$\Xi;\Gamma.\Delta[\mathbf{v}]\vdash$
$\Theta;\Delta\vdash T\ \mathsf{sort}$	$\Theta;\Gamma\vdash T[\vec{v}]\ \mathsf{sort}$	$\Xi;\Gamma.\Delta[\mathbf{v}]\vdash T[\mathbf{v}]\ \mathsf{sort}$
$\Theta;\Delta\vdash t:T$	$\Theta;\Gamma\vdash t[\vec{v}]:T[\vec{v}]$	$\Xi;\Gamma.\Delta[\mathbf{v}]\vdash t[\mathbf{v}]:T[\mathbf{v}]$
$\Theta;\Delta\vdash\vec{t}:\Delta'$	$\Theta;\Gamma\vdash\vec{t}[\vec{v}]:\Delta'$	$\Xi;\Gamma.\Delta[\mathbf{v}]\vdash\mathsf{id}_\Gamma,\vec{t}[\mathbf{v}]:\Gamma.\Delta'[\mathbf{v}]$
$\Theta;\Delta\vdash\mathbf{t}:\Xi'$	$\Theta;\Gamma\vdash\mathbf{t}[\vec{v}]:\Xi'$	$\Xi;\Gamma.\Delta[\mathbf{v}]\vdash\mathbf{t}[\mathbf{v}]:\Xi'$

Proposition 4 (Substitution property). *The following rules are admissible.*

$$\frac{\Theta;\Gamma\vdash\vec{v}:\Delta\qquad\Theta;\Delta\vdash\mathcal{J}}{(\Theta;\Gamma\vdash\mathcal{J})[\vec{v}]}\qquad\qquad\frac{\Xi;\Gamma\vdash\mathbf{v}:\Theta\qquad\Theta;\Delta\vdash\mathcal{J}}{(\Theta;\Delta\vdash\mathcal{J})[\mathbf{v}]}$$

Proof. We illustrate the main case of the second statement's proof, which is by induction on $\Theta;\Delta\vdash\mathcal{J}$. Suppose the derivation ends with the rule MVar.

$$\mathsf{x}\{\Delta'\}:T\in\Theta\;\frac{\Theta;\Delta\vdash\vec{t}:\Delta'}{\Theta;\Delta\vdash\mathsf{x}\{\vec{t}\}:T[\vec{t}]}$$

By i.h. we have $\Xi;\Gamma.\Delta[\mathbf{v}]\vdash\mathsf{id}_\Gamma,\vec{t}[\mathbf{v}]:\Gamma.\Delta'[\mathbf{v}]$. Moreover, from $\Xi;\Gamma\vdash\mathbf{v}:\Theta$ we can derive $\Xi;\Gamma.\Delta'[\mathbf{v}]\vdash\mathbf{v}_\mathsf{x}:T[\mathbf{v}]$, so by the first statement (the substitution property for variable substitutions) we get $\Xi;\Gamma.\Delta[\mathbf{v}]\vdash\mathbf{v}_\mathsf{x}[\mathsf{id}_\Gamma,\vec{t}[\mathbf{v}]]:T[\mathbf{v}][\mathsf{id}_\Gamma,\vec{t}[\mathbf{v}]]$. Because $\mathsf{x}\{\vec{t}\}[\mathbf{v}]=\mathbf{v}_\mathsf{x}[\mathsf{id}_\Gamma,\vec{t}[\mathbf{v}]]$ and $T[\vec{t}][\mathbf{v}]=T[\mathbf{v}][\mathsf{id}_\Gamma,\vec{t}[\mathbf{v}]]$ we are done.

Proposition 5 (Sorts are well-typed). *The following rule is admissible when the underlying theory is valid.*

$$\frac{\Theta; \Gamma \vdash t : T}{\Theta; \Gamma \vdash T \ \mathsf{sort}}$$

Proof. By case analysis on $\Theta; \Gamma \vdash t : T$, and using Proposition 4. For rules Cons and Dest we use the validity of the theory to deduce $\Xi_1 \vdash T \ \mathsf{sort}$ from $c(\Xi_1; \Xi_2) : T \in \mathbb{T}$ and $\Xi_1.(\mathsf{x} : T).\Xi_2 \vdash U \ \mathsf{sort}$ from $d(\Xi_1; \mathsf{x} : T; \Xi_2) : U \in \mathbb{T}$.

Proposition 6 (Conversion in context). *The following rule is admissible.*

$$\Gamma \equiv \Delta \frac{\Theta; \Gamma \vdash \mathcal{J} \qquad \Theta; \Delta \vdash}{\Theta; \Delta \vdash \mathcal{J}}$$

3.4 Subject reduction

A key property that all of our valid theories satisfy is subject reduction. Aside from ensuring that well-typed programs cannot go wrong, this property will be vital to establish the soundness of the bidirectional type system.

In order to show subject reduction, we first need to prove some important properties of patterns. The first of them is injectivity with respect to conversion.

Lemma 1 (Injectivity of patterns). *If $t \in \mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma$ and $t[\mathbf{v}] \equiv t[\mathbf{v}']$ for some $\mathbf{v} \in \mathsf{MSub}\ \theta'\ \delta\ \theta$ and $\mathbf{v}' \in \mathsf{MSub}\ \theta'\ \delta\ \theta$ then $\mathbf{v} \equiv \mathbf{v}'$.*

Proof. By induction on the pattern, generalizing the statement also to metavariable substitution patterns. The key step is in case $c(\mathbf{t}[\mathbf{v}]) \equiv c(\mathbf{t}[\mathbf{v}'])$ in which we crucially rely on Proposition 1 to get $\mathbf{t}[\mathbf{v}] \equiv \mathbf{t}[\mathbf{v}']$ and invoke the i.h. to conclude.

Given a well-typed term t such that the result of substituting $\mathbf{v}$ in t is also well-typed, generally we cannot conclude that the metavariable substitution $\mathbf{v}$ is well-typed. This reasoning is however valid when t is a pattern, as shown by the following result. Differently from the previous lemma, its proof is more intricate so instead of trying to give a proof sketch we refer to the technical report [24].

Proposition 7 (Inversion of pattern typing). *Let $\mathbf{v} \in \mathsf{MSub}\ (\cdot)\ |\Delta|\ |\Theta|$.*

- *If $t \in \mathsf{Tm}^{\mathsf{P}}\ |\Theta|\ (\cdot)$ and $\Theta \vdash t : T$ and $\Delta \vdash t[\mathbf{v}] : T[\mathbf{v}]$ then $\Delta \vdash \mathbf{v} : \Theta$*
- *If $T \in \mathsf{Tm}^{\mathsf{P}}\ |\Theta|\ (\cdot)$ and $\Theta \vdash T\ \mathsf{sort}$ and $\Delta \vdash T[\mathbf{v}]\ \mathsf{sort}$ then $\Delta \vdash \mathbf{v} : \Theta$*
- *If $\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ |\Theta|\ (\cdot)\ |\Xi|$ and $\Theta \vdash \mathbf{t} : \Xi$ and $\Delta \vdash \mathbf{t}[\mathbf{v}] : \Xi$ then $\Delta \vdash \mathbf{v} : \Theta$*

We are now ready to show subject reduction.

Theorem 1 (Subject reduction). *Suppose that the underlying theory is valid.*

- *If $\Gamma \vdash T\ \mathsf{sort}$ and $T \longrightarrow T'$ then $\Gamma \vdash T'\ \mathsf{sort}$*
- *If $\Gamma \vdash t : T$ and $t \longrightarrow t'$ then $\Gamma \vdash t' : T$*
- *If $\Gamma \vdash \mathbf{t} : \Xi$ and $\Xi \vdash$ and $\mathbf{t} \longrightarrow \mathbf{t}'$ then $\Gamma \vdash \mathbf{t}' : \Xi$*

Proof. By induction on the rewrite judgment. We show only the main case:

$$\frac{\theta_1;\theta_2 \Vdash d(t;\mathbf{u}) \longmapsto r \qquad \mathbf{v}_1 \in \mathsf{MSub}\ (\cdot)\ |\Gamma|\ \theta_1 \qquad \mathbf{v}_2 \in \mathsf{MSub}\ (\cdot)\ |\Gamma|\ \theta_2}{d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) \longrightarrow r[\mathbf{v}_1,\mathbf{v}_2]}$$

Let $d(\Xi_1;\mathsf{x}:U;\Xi_2) : V \in \mathbb{T}$ be the rule for d in $\mathbb{T}$. Because $\mathbb{T}$ is valid, there are Θ_1 and Θ_2 with $|\Theta_1| = \theta_1$ and $|\Theta_2| = \theta_2$ such that

$$\Xi_1.\Theta_1.\Theta_2 \vdash (\mathsf{id}_{\Xi_1}, t, \mathbf{u}) : \Xi_1.(\mathsf{x}:U).\Xi_2 \qquad \text{and} \qquad \Xi_1.\Theta_1.\Theta_2 \vdash r : V[\mathsf{id}_{\Xi_1}, t, \mathbf{u}]$$

By inversion on $\Gamma \vdash d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) : T$ we get

$$T \equiv V[\mathbf{v}_0, t[\mathbf{v}_1], \mathbf{u}[\mathbf{v}_2]]\ \frac{\dfrac{\Gamma \vdash \mathbf{v}_0, t[\mathbf{v}_1], \mathbf{u}[\mathbf{v}_2] : \Xi_1.(\mathsf{x}:U).\Xi_2}{\Gamma \vdash d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) : V[\mathbf{v}_0, t[\mathbf{v}_1], \mathbf{u}[\mathbf{v}_2]]}}{\Gamma \vdash d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) : T}$$

Therefore, we have $\Gamma \vdash (\mathsf{id}_{\Xi_1}, t, \mathbf{u})[\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2] : \Xi_1.(\mathsf{x}:U).\Xi_2$, and because $\mathsf{id}_{\Xi_1}, t, \mathbf{u} \in \mathsf{MSub}^{\mathsf{P}}\ |\Xi_1.\Theta_1.\Theta_2|\ (\cdot)\ |\Xi_1.(\mathsf{x}:U).\Xi_2|$, then by Proposition 7 we get $\Gamma \vdash \mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2 : \Xi_1.\Theta_1.\Theta_2$. By applying Proposition 4 with $\Xi_1.\Theta_1.\Theta_2 \vdash r : V[\mathsf{id}_{\Xi_1}, t, \mathbf{u}]$ we get $\Gamma \vdash r[\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2] : V[\mathsf{id}_{\Xi_1}, t, \mathbf{u}][\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2]$. Finally, we have $\Gamma \vdash T\ \mathsf{sort}$ by Proposition 5 applied to $\Gamma \vdash d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) : T$, and because $r[\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2] = r[\mathbf{v}_1,\mathbf{v}_2]$ and $V[\mathsf{id}_{\Xi_1}, t, \mathbf{u}][\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2] = V[\mathbf{v}_0, t[\mathbf{v}_1], \mathbf{u}[\mathbf{v}_2]] \equiv T$ then by the conversion rule we conclude $\Gamma \vdash r[\mathbf{v}_1,\mathbf{v}_2] : T$.

4 Bidirectional type system

In the previous section we have defined the declarative type system. We now move to the bidirectional type system. We start the section by discussing the problem of matching modulo, which is needed for recovering missing arguments. We then introduce the inferable and checkable terms, for which the bidirectional system is defined. We then give the bidirectional typing rules and prove they are sound and complete with respect to the declarative type system. Finally, we use this equivalence to deduce some important properties of the declarative system.

4.1 Matching modulo rewriting

Suppose we want to type $@(t;u)$ by first inferring the sort of t, yielding T. We know that the sort of the principal argument in the rule for $@$ is the pattern $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$, so we could hope to recover A and B by matching T against this pattern. However, because of the conversion rule, in dependent type theories we cannot expect T to be syntactically equal to an instance of this pattern, but only convertible to it. Therefore, our goal is instead to find A and B satisfying $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))[A, x.B] \equiv T$. This shows that the process of recovering missing arguments in bidirectional typing is actually an instance of *matching modulo rewriting* — a connection that seems not to have been noted before in the literature. This also explains why we were careful in Subsection 2.4 to require the sort of a constructor rule and the sort of the principle argument of a destructor rule to be patterns, as they need to support decidable and unitary matching.

In order to explain how to solve matching modulo problems, let us first recall some concepts about rewriting theory. A *(functional) strategy* $\mathfrak{S}$ [11] is defined by a subrelation $\longrightarrow_{\mathfrak{S}} \subseteq \longrightarrow^+$ which has the same normal forms as $\longrightarrow$ and is functional in the sense that $t \longrightarrow_{\mathfrak{S}} u_1$ and $t \longrightarrow_{\mathfrak{S}} u_2$ imply $u_1 = u_2$. Let $\mathsf{m/o}$ be the *maximal outermost* strategy, which contracts all outermost redexes in one step, and write $t \longrightarrow^{\mathsf{h}}_{\mathsf{m/o}} c(\mathbf{u})$ if $c(\mathbf{u})$ is the first term headed by a constructor to which t reduces by $\longrightarrow^{*}_{\mathsf{m/o}}$.

We can now define in Figure 3 an inference system for matching modulo rewriting, which given a pattern t and a term u tries to compute a metavariable substitution $\mathbf{v}$ such that $t[\mathbf{v}] \equiv u$. Note that the use of a specific rewriting strategy is necessary to ensure the functionality of the inference system.

$$\boxed{t \prec u \rightsquigarrow \mathbf{v} \quad (t \in \mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma;\ \ u \in \mathsf{Tm}\ (\cdot)\ \delta.\gamma;\ \ \mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta)}$$

$$u \longrightarrow^{\mathsf{h}}_{\mathsf{m/o}} c(\mathbf{u}) \frac{\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}}{c(\mathbf{t}) \prec u \rightsquigarrow \mathbf{v}}\ \mathtt{Rigid}_{\prec} \qquad \frac{}{\mathsf{x}\{\mathsf{id}_\gamma\} \prec u \rightsquigarrow \vec{x}_\gamma.u}\ \mathtt{Flex}_{\prec}$$

$$\boxed{\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v} \quad (\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi;\ \ \mathbf{u} \in \mathsf{MSub}\ (\cdot)\ \delta.\gamma\ \xi;\ \ \mathbf{v} \in \mathsf{Sub}\ (\cdot)\ \delta\ \theta)}$$

$$\frac{}{\varepsilon \prec \varepsilon \rightsquigarrow \varepsilon}\ \mathtt{EmptyMSub}_{\prec} \qquad \frac{\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}_1 \qquad t \prec u \rightsquigarrow \mathbf{v}_2}{\mathbf{t}, \vec{x}.t \prec \mathbf{u}, \vec{x}.u \rightsquigarrow \mathbf{v}_1, \mathbf{v}_2}\ \mathtt{ExtMSub}_{\prec}$$

Fig. 3. Inference system for matching modulo

Let us now establish the correctness of this inference system in three steps. We first show its soundness, which follows by an easy induction on the derivation.

Proposition 8 (Soundness of matching).

– *If* $t \prec u \rightsquigarrow \mathbf{v}$ *then* $u \longrightarrow^{*} t[\mathbf{v}]$.
– *If* $\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}$ *then* $\mathbf{u} \longrightarrow^{*} \mathbf{t}[\mathbf{v}]$.

In order to show completeness we first have to answer the following question: if $t \equiv c(\mathbf{u})$, are we sure that by reducing t we eventually reach a term headed by c? The answer would be no, had we taken for instance an innermost strategy. Thankfully, because the rewrite systems of our theories are both orthogonal and *fully-extended*, it follows by [39, Theorem 2] that the maximal-outermost strategy we are using is *head-normalizing*, and so we have the following property.

Lemma 2. *If* $u \equiv c(\mathbf{t})$ *then* $u \longrightarrow^{\mathsf{h}}_{\mathsf{m/o}} c(\mathbf{u})$ *with* $\mathbf{t} \equiv \mathbf{u}$.

Using Lemma 2, we can now show completeness by induction on the pattern.

Proposition 9 (Completeness of matching).

- *If* $t[\mathbf{v}] \equiv u$ *for some* $\mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta$ *then* $t \prec u \rightsquigarrow \mathbf{v}'$ *for some* $\mathbf{v}' \equiv \mathbf{v}$
- *If* $\mathbf{t}[\mathbf{v}] \equiv \mathbf{u}$ *for some* $\mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta$ *then* $\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}'$ *for some* $\mathbf{v}' \equiv \mathbf{v}$

Recall that an expression is weak normalizing if it reduces to a normal form. We can now show that the inference system is decidable when being used with weak normalizing expressions. The proof is by induction on the pattern, using the fact that $\mathsf{m/o}$ is *normalizing* [42, Theorem 10], so that reducing a weak normalizing term with $\mathsf{m/o}$ always terminates.

Proposition 10 (Decidability of matching).

- *If* u *is weak normalizing, then* $\exists \mathbf{v}.\ t \prec u \rightsquigarrow \mathbf{v}$ *is decidable for all* t.
- *If* $\mathbf{u}$ *is weak normalizing, then* $\exists \mathbf{v}.\ \mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}$ *is decidable for all* $\mathbf{t}$.

4.2 Inferable and checkable terms

Before giving the bidirectional typing rules, we first have to address the problem that some terms without annotations cannot be algorithmically typed. Suppose for instance that we want to type the term $@(\lambda(x.t); u)$ by inferring the sort of the principal argument of $@$ to recover A and B. But because $\lambda(x.t)$ is headed by a constructor it can only be bidirectionally typed in mode check, so we are stuck. One could think that this limitation is specific to bidirectional typing, however a famous result by Dowek shows that, in a dependently-typed setting, the problem of typing unannotated terms is actually undecidable in its full generality [18].

To solve this issue, we have two options. We could proceed as in the Coq literature [33] and add extra annotations to terms so that they can always be inferred. For instance, we would then need to annotate abstraction with its domain by writing $\lambda(A, x.t)$ instead of $\lambda(x.t)$. However, not only this solution makes the syntax heavier, but by abandoning the constructor/destructor separation in which constructors are always typed in mode check, it yields typing rules which are much less symmetric and whose form seems difficult to specify in a generic way.

We instead take the choice made by most of the dependent bidirectional typing literature [2,38,1,16,3,4] and define our bidirectional system only for a subset of expressions, the *checkable and inferable terms* and the *checkable metavariable substitutions*. Given a fixed signature Σ, they are defined as follows.

$$\begin{array}{rlll}
\boxed{\mathsf{Tm}^{\mathsf{i}}\ \gamma} \ni & t, u, v ::= & \mid x & \text{if } x \in \gamma \\
& & \mid d(t \in \mathsf{Tm}^{\mathsf{i}}\ \gamma; \mathbf{t} \in \mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi) & \text{if } d(\xi) \in \Sigma \\
\boxed{\mathsf{Tm}^{\mathsf{c}}\ \gamma} \ni & t, u, v ::= & \mid c(\mathbf{t} \in \mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi) & \text{if } c(\xi) \in \Sigma \\
& & \mid \underline{t} \in \mathsf{Tm}^{\mathsf{i}}\ \gamma & \\
\boxed{\mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi} \ni & \mathbf{t}, \mathbf{u}, \mathbf{v} ::= & \mid \varepsilon & \text{if } \xi = \cdot \\
& & \mid \mathbf{t}' \in \mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{Tm}^{\mathsf{c}}\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\}
\end{array}$$

Let us go through the definition. An inferable term is either a variable or a destructor whose principal argument is inferable, and whose other arguments are given by a checkable metavariable substitution. Imposing the principal argument to be inferable is the key restriction to rule out terms like $@(\lambda(x.t); u)$. A checkable term is then either a constructor applied to a checkable metavariable substitution, or an underlined inferable term. Finally, a checkable metavariable substitution is just a metavariable substitution containing only checkable terms.

Example 9. The inferrable and checkable terms for the signature $\Sigma_{\lambda\Pi}$ are given respectively by the following grammars, where we omit the scope information.

$$t^{\mathsf{i}}, u^{\mathsf{i}} ::= x \mid @(t^{\mathsf{i}}; u^{\mathsf{c}})$$
$$t^{\mathsf{c}}, u^{\mathsf{c}}, A^{\mathsf{c}}, B^{\mathsf{c}} ::= \mathrm{Ty} \mid \mathrm{Tm}(A^{\mathsf{c}}) \mid \Pi(A^{\mathsf{c}}, x.B^{\mathsf{c}}) \mid \lambda(x.t^{\mathsf{c}}) \mid \underline{t^{\mathsf{i}}}$$

One could wonder if restricting the terms that can be algorithmically typed is a significant limitation. For most usual theories (like $\mathbb{T}_{\lambda\Pi}$ and those in Section 5) the checkable terms coincide with the normal forms, and the inferable terms coincide with the neutrals. As argued in other works [38,1], users of type theory almost only write terms in normal form, and in the few cases writing a redex is more convenient, in actual implementations the principal argument can always be lifted to a top-level definition. Therefore, this restriction, also present in most of the dependent bidirectional typing literature [2,38,1,16,3,4], does not pose a serious limitation in practice.

Note also that inferable and checkable terms have no metavariables. Even if metavariables are needed in the declarative system to be able to say which theories are valid, when writing terms in a fixed theory metavariables are generally not needed, and hence they are omitted from usual presentations of type theories. It is therefore reasonable to leave them out of the bidirectional syntax, as they will be of no use for users.

Given $t \in \mathsf{Tm}^{\mathsf{c}}\ \gamma$ or $t \in \mathsf{Tm}^{\mathsf{i}}\ \gamma$ we write $\ulcorner t \urcorner \in \mathsf{Tm}\ (\cdot)\ \gamma$ for its underlying term, and for $\mathbf{t} \in \mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi$ we also write $\ulcorner \mathbf{t} \urcorner \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ for its underlying metavariable substitution.

4.3 Bidirectional typing rules

Given a theory $\mathbb{T}$, we can now define its bidirectional type system by the rules in Figure 4. The system is split in 4 judgments:

- $\Gamma \vdash T \Leftarrow \mathsf{sort}$: Checking that a checkable term T is a well-formed sort
- $\Gamma \vdash t \Leftarrow T$: Checking that a checkable term t has sort T
- $\Gamma \vdash t \Rightarrow T$: Inferring a sort T for an inferable term t
- $\Gamma \mid \mathbf{v} : \Xi \vdash \mathbf{t} \Leftarrow \Theta$: Checking that a checkable metavariable substitution $\mathbf{t}$ can be typed by Θ "to the right" of $\mathbf{v} : \Xi$

As in the declarative system, the most important rules are the one that instantiate the schematic typing rules: Const, Dest and Sort. However, differently from the declarative system, no more guessing is needed when building

$$\boxed{\Gamma \vdash t \Leftarrow T \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{Tm}\ (\cdot)\ |\Gamma|;\ t \in \mathsf{Tm}^{\mathsf{c}}\ |\Gamma|)}$$

$$c(\Xi_1;\Xi_2) : T \in \mathbb{T}\ \dfrac{T \prec U \rightsquigarrow \mathbf{v} \qquad \Gamma \mid \mathbf{v} : \Xi_1 \vdash \mathbf{u} \Leftarrow \Xi_2}{\Gamma \vdash c(\mathbf{u}) \Leftarrow U}\,\textsc{Cons} \qquad T \equiv_{\mathbb{T}} U\ \dfrac{\Gamma \vdash t \Rightarrow T}{\Gamma \vdash \underline{t} \Leftarrow U}\,\textsc{Switch}$$

$$\boxed{\Gamma \vdash t \Rightarrow T \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{Tm}\ (\cdot)\ |\Gamma|;\ t \in \mathsf{Tm}^{\mathsf{i}}\ |\Gamma|)}$$

$$d(\Xi_1;\mathsf{x}:T;\Xi_2) : U \in \mathbb{T}\ \dfrac{\begin{array}{c}\Gamma \vdash t \Rightarrow V \qquad T \prec V \rightsquigarrow \mathbf{v} \\ \Gamma \mid (\mathbf{v}, \ulcorner t \urcorner) : (\Xi_1, \mathsf{x}:T) \vdash \mathbf{u} \Leftarrow \Xi_2\end{array}}{\Gamma \vdash d(t;\mathbf{u}) \Rightarrow U[\mathbf{v}, \ulcorner t \urcorner, \ulcorner \mathbf{u} \urcorner]}\,\textsc{Dest} \qquad x : T \in \Gamma\ \dfrac{}{\Gamma \vdash x \Rightarrow T}\,\textsc{Var}$$

$$\boxed{\Gamma \vdash T \Leftarrow \mathsf{sort} \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{Tm}^{\mathsf{c}}\ |\Gamma|)}$$

$$c(\Xi)\ \mathsf{sort} \in \mathbb{T}\ \dfrac{\Gamma \mid \varepsilon : (\cdot) \vdash \mathbf{t} \Leftarrow \Xi}{\Gamma \vdash c(\mathbf{t}) \Leftarrow \mathsf{sort}}\,\textsc{Sort}$$

$$\boxed{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi \quad \left(\begin{array}{c}\Gamma \in \mathsf{Ctx};\ \Theta \in \mathsf{MCtx};\ \mathbf{v} \in \mathsf{MSub}\ (\cdot)\ |\Gamma|\ |\Theta|; \\ \Xi \in \mathsf{MCtx}\ |\Theta|;\ \mathbf{t} \in \mathsf{MSub}^{\mathsf{c}}\ |\Gamma|\ |\Xi|\end{array}\right)}$$

$$\dfrac{}{\Gamma \mid \mathbf{v} : \Theta \vdash \varepsilon \Leftarrow (\cdot)}\,\textsc{EmptyMSub} \qquad \dfrac{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi \qquad \Gamma.\Delta[\mathbf{v}, \ulcorner \mathbf{t} \urcorner] \vdash t \Leftarrow T[\mathbf{v}, \ulcorner \mathbf{t} \urcorner]}{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t}, \vec{x}_\Delta.t \Leftarrow (\Xi, \mathsf{x}\{\Delta\} : T)}\,\textsc{ExtMSub}$$

Fig. 4. Bidirectional typing rules

a type derivation. For instance, when using rule Dest with $d(t;\mathbf{u})$ the omitted arguments are no longer guessed, but instead recovered by inferring the sort of the principal argument t and then matching it against the associated pattern.

Example 10. Suppose we want to infer a sort for $@(t;u)$ in the theory $\mathbb{T}_{\lambda\Pi}$. To use rule Dest, we start by inferring a sort V for t, and then we try to match it against the pattern $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$. If matching succeeds, we recover the arguments A and B, which together with t are then used in

$$\Gamma \mid (A, x.B, x.\ulcorner t \urcorner) : (\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty},\ \mathtt{t} : ...) \vdash (u) \Leftarrow (\mathsf{u} : \mathrm{Tm}(\mathsf{A}))$$

where we omit the sort of $\mathtt{t}$ for lack of space. By applying the rules that define the judgment $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$, we see that this amounts to showing just $\Gamma \vdash u \Leftarrow \mathrm{Tm}(A)$, and so the final shape of this "big-step derivation" is the following, which corresponds to the usual bidirectional rule for application.

$$\dfrac{\Gamma \vdash t \Rightarrow V \qquad \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \prec V \rightsquigarrow A, x.B \qquad \Gamma \vdash u \Leftarrow \mathrm{Tm}(A)}{\Gamma \vdash @(t;u) \Rightarrow \mathrm{Tm}(B[\mathsf{id}_\Gamma, \ulcorner u \urcorner])}$$

4.4 Equivalence with declarative typing

We now establish the equivalence between the declarative and bidirectional type systems. This is done in two steps, the first one being soundness:

Theorem 2 (Soundness). *Suppose that the underlying theory $\mathbb{T}$ is valid.*

1. *If $\Gamma \vdash$ and $\Gamma \vdash t \Rightarrow T$ then $\Gamma \vdash \ulcorner t \urcorner : T$*
2. *If $\Gamma \vdash T$ sort and $\Gamma \vdash t \Leftarrow T$ then $\Gamma \vdash \ulcorner t \urcorner : T$*
3. *If $\Gamma \vdash$ and $\Gamma \vdash T \Leftarrow \mathsf{sort}$ then $\Gamma \vdash \ulcorner T \urcorner$ sort*
4. *If $\Gamma \vdash \mathbf{v} : \Xi_1$ and $\Xi_1.\Xi_2 \vdash$ and $\Gamma \mid \mathbf{v} : \Xi_1 \vdash \mathbf{t} \Leftarrow \Xi_2$ then $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Xi_1.\Xi_2$.*

Proof. By induction on the derivation. We illustrate one of the interesting cases.

$$c(\Xi_1;\Xi_2) : T \in \mathbb{T} \; \frac{T \prec U \rightsquigarrow \mathbf{v} \qquad \Gamma \mid \mathbf{v} : \Xi_1 \vdash \mathbf{u} \Leftarrow \Xi_2}{\Gamma \vdash c(\mathbf{u}) \Leftarrow U}$$

By Proposition 8 we have $U \longrightarrow^* T[\mathbf{v}]$, so because we have $\Gamma \vdash U$ sort then by Theorem 1 we get $\Gamma \vdash T[\mathbf{v}]$ sort. We have $T \in \mathsf{Tm}^{\mathsf{P}}$ $|\Xi_1|$ $(\cdot)$, and validity of the theory also gives $\Xi_1 \vdash T$ sort, therefore by Proposition 7 we get $\Gamma \vdash \mathbf{v} : \Xi_1$. By validity of the theory we also have $\Xi_1.\Xi_2 \vdash$, therefore by applying the i.h. to the second premise we get $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{u} \urcorner : \Xi_1.\Xi_2$. We can then derive $\Gamma \vdash c(\ulcorner \mathbf{u} \urcorner) : T[\mathbf{v}]$, and because $T[\mathbf{v}] \equiv U$ and $\Gamma \vdash U$ sort, by conversion we conclude $\Gamma \vdash c(\ulcorner \mathbf{u} \urcorner) : U$.

Completeness then asserts that checkable/inferable terms typable in the declarative system can also be typed in the bidirectional system.

Theorem 3 (Completeness). *Suppose that the underlying theory $\mathbb{T}$ is valid.*

1. *If t is inferable and $\Gamma \vdash \ulcorner t \urcorner : T$ then $\Gamma \vdash t \Rightarrow T'$ with $T \equiv T'$*
2. *If t is checkable and $\Gamma \vdash \ulcorner t \urcorner : T$ then we have $\Gamma \vdash t \Leftarrow T$*
3. *If T is checkable and $\Gamma \vdash \ulcorner T \urcorner$ sort then $\Gamma \vdash T \Leftarrow \mathsf{sort}$*
4. *If $\mathbf{t}$ is checkable and $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Theta.\Xi$ then we have $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$*

Proof. The proof requires us to strengthen the statement, so the two occurrences of the context Γ in points 1-4, of the sort T in point 2 and of the substitution $\mathbf{v}$ in point 4 are not required to be syntactically equal, but only convertible (see the technical report [24] for the exact statement). The proof is then by induction on the checkable/inferable term or checkable substitution.

We illustrate the case $t = d(u; \mathbf{t})$, in which we have to show that for all $\Gamma' \equiv \Gamma$ we have some $T' \equiv T$ such that $\Gamma' \vdash t \Rightarrow T'$. By inversion on $\Gamma \vdash \ulcorner t \urcorner : T$ we have

$$\cfrac{d(\Xi_1; \mathsf{x} : U; \Xi_2) : V \in \mathbb{T} \; \cfrac{\Gamma \vdash \mathbf{v}, \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner : \Xi_1.(\mathsf{x} : U).\Xi_2}{\Gamma \vdash d(\ulcorner u \urcorner; \ulcorner \mathbf{t} \urcorner) : V[\mathbf{v}, \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner]} \quad T \equiv V[\mathbf{v}, \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner]}{\Gamma \vdash d(\ulcorner u \urcorner; \ulcorner \mathbf{t} \urcorner) : T}$$

Let $\Gamma' \equiv \Gamma$. From $\Gamma \vdash \mathbf{v}, \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner : \Xi_1.(\mathsf{x} : U).\Xi_2$ we get $\Gamma \vdash \ulcorner u \urcorner : U[\mathbf{v}]$, so because u is inferrable, by the i.h. we obtain that for some $U' \equiv U[\mathbf{v}]$ we have $\Gamma' \vdash u \Rightarrow U'$. By Proposition 9 we then get $U \prec U' \rightsquigarrow \mathbf{v}'$ with $\mathbf{v} \equiv \mathbf{v}'$. Then, because $\mathbf{t}$ is checkable and $\Gamma \equiv \Gamma'$ and $\mathbf{v}, \ulcorner u \urcorner \equiv \mathbf{v}', \ulcorner u \urcorner$, by the i.h. we derive $\Gamma' \mid (\mathbf{v}', \ulcorner u \urcorner) : (\Xi_1, \mathsf{x} : U) \vdash \mathbf{t} \Leftarrow \Xi_2$. Putting all this together, we conclude $\Gamma' \vdash d(u; \mathbf{t}) \Rightarrow V[\mathbf{v}', \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner]$, where we have $V[\mathbf{v}', \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner] \equiv T$ as required.

4.5 Consequences of the equivalence

We now explore the established equivalence in order to show two important properties: decidability of typing and uniqueness of sorts.

We say that a theory $\mathbb{T}$ is weak normalizing if for all expressions e with $\Gamma \vdash e\ \mathsf{sort}$ or $\Gamma \vdash e : T$ or $\Gamma \vdash e : \Xi$ we have that e is weak normalizing.

Theorem 4 (Decidability of typing). *Suppose that the underlying theory $\mathbb{T}$ is valid and weak normalizing.*

1. *If t is inferable and $\Gamma \vdash$ then the statement $\exists T.\ (\Gamma \vdash \ulcorner t \urcorner : T)$ is decidable.*
2. *If t is checkable and $\Gamma \vdash T\ \mathsf{sort}$ then the statement $\Gamma \vdash \ulcorner t \urcorner : T$ is decidable.*
3. *If T is checkable and $\Gamma \vdash$ then the statement $\Gamma \vdash \ulcorner T \urcorner\ \mathsf{sort}$ is decidable.*
4. *If $\mathbf{t}$ is checkable and $\Theta.\Xi \vdash$ and $\Gamma \vdash \mathbf{v} : \Theta$ then the statement $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Theta.\Xi$ is decidable.*

Proof. We first show the corresponding statement for the bidirectional system, using Proposition 10, Theorem 2 and the decidability of conversion for well-typed terms (which follows from weak normalization). By Theorems 2 and 3 we can then conclude. We refer to the technical report [24] for the proof.

We now move to uniqueness of sorts. First note that, because our terms are non-annotated, uniqueness of sorts does not hold in general: for instance, $\lambda(x.x)$ can be typed by $\mathrm{Tm}(\Pi(A, x.A))$ in context Γ for any A with $\Gamma \vdash A : \mathrm{Ty}$. Nevertheless, we can still show uniqueness of sorts for inferable terms:

Theorem 5 (Uniqueness of sorts). *Suppose that the underlying theory $\mathbb{T}$ is valid. If t is inferable and $\Gamma \vdash \ulcorner t \urcorner : T$ and $\Gamma \vdash \ulcorner t \urcorner : U$ then $T \equiv U$.*

Proof. By Theorem 3 we get $\Gamma \vdash t \Rightarrow T'$ with $T \equiv T'$ from $\Gamma \vdash \ulcorner t \urcorner : T$ and $\Gamma \vdash t \Rightarrow U'$ with $U \equiv U'$ from $\Gamma \vdash \ulcorner t \urcorner : U$. We can show type inference to be functional, so we get $T' = U'$ and thus $T \equiv U$.

5 More examples

In the previous sections we have illustrated our framework with the theory $\mathbb{T}_{\lambda\Pi}$, defining a basic Martin-Löf Type Theory with dependent products. We now show other examples of valid theories to showcase the generality of our framework. Throughout this section, we use the informal notation for schematic typing rules discussed in Subsection 2.4, for readability purposes. We refer to the files of the implementation [23] for more details.

Lists We can define lists by extending $\mathbb{T}_{\lambda\Pi}$ with the following.

$$\frac{\vdash \mathsf{A} : \mathrm{Ty}}{\vdash \mathrm{List}(\mathsf{A}) : \mathrm{Ty}} \qquad \frac{\vdash \mathsf{A} : \mathrm{Ty}}{\vdash \mathrm{nil} : \mathrm{Tm}(\mathrm{List}(\mathsf{A}))} \qquad \frac{\vdash \mathsf{A} : \mathrm{Ty} \qquad \vdash \mathsf{x} : \mathrm{Tm}(\mathsf{A}) \\ \vdash \mathsf{l} : \mathrm{Tm}(\mathrm{List}(\mathsf{A}))}{\vdash \mathrm{cons}(\mathsf{x}, \mathsf{l}) : \mathrm{Tm}(\mathrm{List}(\mathsf{A}))}$$

$$\frac{\vdash \mathsf{A} : \mathrm{Ty} \qquad \vdash \mathsf{l} : \mathrm{Tm}(\mathrm{List}(\mathsf{A})) \qquad x : \mathrm{Tm}(\mathrm{List}(\mathsf{A})) \vdash \mathsf{P} : \mathrm{Ty} \qquad \vdash \mathsf{pnil} : \mathrm{Tm}(\mathsf{P}\{\mathrm{nil}\}) \\ x : \mathrm{Tm}(\mathsf{A}), y : \mathrm{Tm}(\mathrm{List}(\mathsf{A})), z : \mathrm{Tm}(\mathsf{P}\{y\}) \vdash \mathsf{pcons} : \mathrm{Tm}(\mathsf{P}\{\mathrm{cons}(x, y)\})}{\vdash \mathrm{ListRec}(\mathsf{l}; \mathsf{P}, \mathsf{pnil}, \mathsf{pcons}) : \mathrm{Tm}(\mathsf{P}\{\mathsf{l}\})}$$

$$\mathrm{ListRec}(\mathrm{nil}; x.\mathsf{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x,y,z\}) \longmapsto \mathtt{pnil}$$

$$\mathrm{ListRec}(\mathrm{cons}(\mathsf{x},\mathtt{l}); x.\mathsf{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x,y,z\}) \longmapsto$$
$$\mathtt{pcons}\{\mathsf{x}, \mathtt{l}, \mathrm{ListRec}(\mathtt{l}; x.\mathsf{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x,y,z\})\}$$

Like one would wish, the constructors nil and cons indeed do not store the type annotation A, which is recovered from the sort. This annotation is also elided in the destructor ListRec, where it is recovered from the principal argument.

In general, we can extend the theory with any (non-indexed) inductive type. For instance, see the file `mltt.bitt` where we add dependent sums and W types.

Universes We can define *Tarski-style* universes by extending $\mathbb{T}_{\lambda\Pi}$ with a type U and a decoding function mapping each inhabitant a of U into a type $\mathrm{El}(a;\varepsilon)$.

$$\frac{}{\vdash \mathrm{U} : \mathrm{Ty}} \qquad \frac{\vdash \mathsf{a} : \mathrm{Tm}(\mathrm{U})}{\vdash \mathrm{El}(\mathsf{a};\cdot) : \mathrm{Ty}}$$

We then add a code for each type of the theory, with an associated rewriting rule stating that the code is decoded by El into the appropriate type.

$$\frac{}{\vdash \mathrm{u} : \mathrm{Tm}(\mathrm{U})} \qquad \frac{\vdash \mathsf{a} : \mathrm{Tm}(\mathrm{U}) \qquad x : \mathrm{Tm}(\mathrm{El}(\mathsf{a};\varepsilon)) \vdash \mathsf{b} : \mathrm{Tm}(\mathrm{U})}{\vdash \pi(\mathsf{a},\mathsf{b}) : \mathrm{Tm}(\mathrm{U})}$$

$$\mathrm{El}(\mathrm{u};\varepsilon) \longmapsto \mathrm{U} \qquad \mathrm{El}(\pi(\mathsf{a}, x.\mathsf{b}\{x\});\varepsilon) \longmapsto \Pi(\mathrm{El}(\mathsf{a};\varepsilon), x.\mathrm{El}(\mathsf{b}\{x\};\varepsilon))$$

This specifies a type in type universe, which is known to be inconsistent [15]. This can however be easily fixed by stratifying universes into a hierarchy. By doing this, we can then define a Tarski-style variant of (functional) *Pure Type Systems* [9]. Alternatively, instead of using Tarski-style we can also define (weak) *Coquand-style universes* [17,30,32,6] which require replacing the sorts Ty and Tm by indexed families Ty_i and Tm_i — see the file `mltt-coquand.bitt` for a definition also featuring (weak) cumulativity and universe polymorphism.

Higher-order logic We have seen how to extend $\mathbb{T}_{\lambda\Pi}$ with various type formers, however we can also define logics. To define higher-order logic (HOL) we first declare a type of propositions and a sort rule to represent the judgment "P is true".

$$\frac{}{\vdash \mathrm{Prop} : \mathrm{Ty}} \qquad \frac{\vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop})}{\vdash \mathrm{Prf}(\mathsf{P})\ \mathsf{sort}}$$

We can then add connectors or quantifiers such as the universal quantifier $\forall$ — see the file `hol.bitt` for more details.

$$\frac{\begin{array}{c}\vdash \mathsf{A} : \mathrm{Ty} \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop})\end{array}}{\vdash \forall(\mathsf{A},\mathsf{P}) : \mathrm{Tm}(\mathrm{Prop})} \qquad \frac{\begin{array}{c}\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop}) \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{p} : \mathrm{Prf}(\mathsf{P})\end{array}}{\vdash \forall_{\mathrm{in}}(\mathsf{p}) : \mathrm{Prf}(\forall(\mathsf{A}, x.\mathsf{P}\{x\}))}$$

$$\frac{\begin{array}{c}\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop}) \\ \vdash \mathsf{r} : \mathrm{Prf}(\forall(\mathsf{A}, x.\mathsf{P}\{x\})) \qquad \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{A})\end{array}}{\vdash \forall_{\mathrm{el}}(\mathsf{r};\mathsf{t}) : \mathrm{Prf}(\mathsf{P}\{\mathsf{t}\})} \qquad \forall_{\mathrm{el}}(\forall_{\mathrm{in}}(x.\mathsf{p}\{x\});\mathsf{t}) \longmapsto \mathsf{p}\{\mathsf{t}\}$$

6 Related work

Our general definition of dependent type theories draws much inspiration from other frameworks for type theory, such as GATs/QIITs [13,7,31], SOGATs [45], FTTs [29], and logical frameworks such as Dedukti [8,12] and Harper's Equational LF [27]. However, we differ from these works by supporting non-annotated syntaxes and enforcing a constructor/destructor separation of symbols and rules, both of which seem to be important ingredients for bidirectional typing.

Another point of divergence from these frameworks is that most of them allow the use of arbitrary equations when defining the definitional equality of theories. However, it then becomes hard to give an implementation, as it would require deciding arbitrary equational theories. We instead take the approach of Dedukti of supporting only rewrite rules, which allows to decide the definitional equality of theories in a uniform manner, and makes it possible to implement our framework. A different approach is taken in Andromeda, an implementation of FTTs, where they also allow for extensionality rules [10]. They however provide no proof of completeness for their equality-checking algorithm.

Our proposal also draws inspiration from the works of McBride, a main advocate of dependent bidirectional typing. His ongoing work on a framework for bidirectional typing [35,36] shares many similarities with ours, for instance by adopting a constructor/destructor separation of rules. However, an important difference with our framework is that he takes the bidirectional type system as the definition of the theory. Therefore, there is no discussion on how to show soundness and completeness with respect to a declarative system, as the bidirectional one is the only type system defined in his setting. This approach differs from most of the literature on dependent bidirectional typing [19,2,33,1,3,4], in which one first defines the type theory by a "platonic" declarative type system and then shows it equivalent to a bidirectional system which can be implemented. Finally, this choice also makes the metatheoretic study of theories quite different: for instance, in order for the bidirectional system to satisfy subject reduction he is obliged to introduce type ascriptions in the syntax.

Another work from which ours drew inspiration is the one of Reed [43], where he proposes a variant of the Edinburgh Logical Framework in which arguments can be omitted. Crucially, these arguments are not elaborated through global unification, but instead locally recovered by annotating each declaration with modes to guide a bidirectional algorithm. However, his framework does not allow for extending the definitional equality, meaning that one cannot define dependent type theories directly, but instead has to encode its derivations trees (as in [28]). This also means that his system does not need to deal with some complications that arise in our more general setting, such as matching modulo.

Finally, concurrently to our work, Chen and Ko [14] have proposed a framework for simply-typed bidirectional typing. They also define declarative and bidirectional systems and establish a correspondence between them. Compared to our work, their restriction to simple types removes many of the complexities that appear in dependent type theories. For instance, while their types are first-order terms with no notion of computation or typing, our sorts are higher-order terms considered modulo a set of rewrite rules and subject to typing judgments, making the process of recovering missing arguments much more intricate. They however formalize all their proofs in Agda.

7 Conclusion and future work

In this work we have given a generic account of bidirectional typing for a general class of dependent type theories. Our main results, Theorems 2 and 3, establish an equivalence between declarative and bidirectional type systems for a general class of theories. The underlying algorithm of Theorem 4, establishing the decidability of typing for weak normalizing theories, has been implemented in a prototype further described in an accompanying experience report [22]. Compared to other theory-independent typecheckers, such as Dedukti, its support for unannotated syntaxes should allow for better performances, which can make it a good candidate for cross-checking real proof libraries.

Regarding future work, the most important omission that we would like to address is that of inductive families. Indeed, these do not fit our definitions because their constructor's sorts either are non-linear patterns (as in the constructor for equality) or contain metavariables for arguments that are computationally relevant and thus cannot be omitted (as in the cons constructor for vectors).

Moreover, even if our system builds heavily on the constructor/destructor distinction in type theory, some few constructions do not respect this separation. For instance, to define Russell universes we need the rewrite rule $\mathrm{Tm}(\mathrm{U}) \longmapsto \mathrm{Ty}$ [44], which is not valid as Tm is not a destructor. Whether there is a way of accommodating these constructions without fully abandoning the constructor/destructor separation is something we would like to investigate in future work.

A long term goal is also to extend our framework to account for type-directed equality rules, which are needed for handling η-laws and definitional proof irrelevance. Even if it is well known how to design complete equality checking algorithms for specific theories with type-directed equalities [5], doing so in a general setting like ours seems to be an important challenge. We could take inspiration from the customizable equality-checking algorithm implemented in Andromeda [10]. However, as mentioned in the previous section, their algorithm is not proven complete, so further research in this direction seems to be needed.

Acknowledgements The author thanks Gilles Dowek, Vincent Moreau, Théo Winterhalter for helpful remarks on a preliminary version of this paper, Meven Lennon-Bertrand, Jesper Cockx, András Kovács for informative discussions about bidirectional typing, and Frédéric Blanqui for valuable remarks about this work.

Data-Availability Statement The implementation associated with this work is available online [23] and is described in an accompanying experience report [22].

References

1. Abel, A., Altenkirch, T.: A partial type checking algorithm for type: Type. Electronic Notes in Theoretical Computer Science **229**(5), 3–17 (2011)
2. Abel, A., Coquand, T.: Untyped algorithmic equality for martin-löf's logical framework with surjective pairs. In: Typed Lambda Calculi and Applications: 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005. Proceedings 7. pp. 23–38. Springer (2005)
3. Abel, A., Coquand, T., Pagano, M.: A modular type-checking algorithm for type theory with singleton types and proof irrelevance. Logical Methods in Computer Science **7** (2011)
4. Abel, A., Vezzosi, A., Winterhalter, T.: Normalization by evaluation for sized dependent types. Proceedings of the ACM on Programming Languages **1**(ICFP), 1–30 (2017)
5. Abel, A., Öhman, J., Vezzosi, A.: Decidability of conversion for type theory in type theory. Proceedings of the ACM on Programming Languages **2**(POPL), 1–29 (Jan 2018). https://doi.org/10.1145/3158111, https://dl.acm.org/doi/10.1145/3158111
6. Altenkirch, T., Boulier, S., Kaposi, A., Tabareau, N.: Setoid type theory—a syntactic translation. In: Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings 13. pp. 155–196. Springer (2019)
7. Altenkirch, T., Kaposi, A.: Type theory in type theory using quotient inductive types. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 18–29. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837638, https://doi.org/10.1145/2837614.2837638
8. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory (2016), unpublished
9. Barendregt, H.P., Dekkers, W., Statman, R.: Lambda calculus with types. Cambridge University Press (2013)
10. Bauer, A., Komel, A.P.: An extensible equality checking algorithm for dependent type theories. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/lmcs-18(1:17)2022, https://doi.org/10.46298/lmcs-18(1:17)2022
11. Bezem, M., Klop, J., de Vrijer, R., Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (2003), https://books.google.fr/books?id=7QQ5u-4tRUkC
12. Blanqui, F., Dowek, G., Grienenberger, É., Hondet, G., Thiré, F.: Some axioms for mathematics. In: Kobayashi, N. (ed.) 6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference). LIPIcs, vol. 195, pp. 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.FSCD.2021.20, https://doi.org/10.4230/LIPIcs.FSCD.2021.20
13. Cartmell, J.: Generalised algebraic theories and contextual categories. Annals of Pure and Applied Logic **32**, 209–243 (1986). https://doi.org/https://doi.org/10.1016/0168-0072(86)90053-9, https://www.sciencedirect.com/science/article/pii/0168007286900539
14. Chen, L.T., Ko, H.S.: A formal treatment of bidirectional typing (2024)
15. Coquand, T.: An analysis of Girard's paradox. Tech. Rep. RR-0531, INRIA (May 1986), https://inria.hal.science/inria-00076023

16. Coquand, T.: An algorithm for type-checking dependent types. Science of Computer Programming **26**(1-3), 167–177 (1996)
17. Coquand, T.: Canonicity and normalisation for dependent type theory. arXiv preprint arXiv:1810.09367 (2018)
18. Dowek, G.: The undecidability of typability in the lambda-pi-calculus. In: International Conference on Typed Lambda Calculi and Applications. pp. 139–145. Springer (1993)
19. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Computing Surveys (CSUR) **54**(5), 1–38 (2021)
20. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 281–292. POPL '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/964001.964025, https://doi.org/10.1145/964001.964025
21. Felicissimo, T.: Adequate and Computational Encodings in the Logical Framework Dedukti. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 228, pp. 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.25, https://drops.dagstuhl.de/opus/volltexte/2022/16306
22. Felicissimo, T.: Artifact report: Generic bidirectional typing for dependent type theories (2024)
23. Felicissimo, T.: BiTTs (Jan 2024). https://doi.org/10.5281/zenodo.10500598, https://doi.org/10.5281/zenodo.10500598
24. Felicissimo, T.: Generic bidirectional typing for dependent type theories (Technical Report) (2024), https://inria.hal.science/hal-04270368/file/tech.pdf
25. Gratzer, D., Sterling, J., Birkedal, L.: Implementing a modal dependent type theory. Proceedings of the ACM on Programming Languages **3**(ICFP), 1–29 (2019)
26. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1), 143–184 (1993)
27. Harper, R.: An equational logical framework for type theories. arXiv preprint arXiv:2106.01484 (2021)
28. Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. Journal of functional programming **17**(4-5), 613–673 (2007)
29. Haselwarter, P.G., Bauer, A.: Finitary type theories with and without contexts. arXiv preprint arXiv:2112.00539 (2021)
30. Kaposi, A., Huber, S., Sattler, C.: Gluing for type theory. In: 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
31. Kaposi, A., Kovács, A., Altenkirch, T.: Constructing quotient inductive-inductive types. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290315, https://doi.org/10.1145/3290315
32. Kovács, A.: Staged compilation with two-level type theory. Proceedings of the ACM on Programming Languages **6**(ICFP), 540–569 (2022)
33. Lennon-Bertrand, M.: Complete Bidirectional Typing for the Calculus of Inductive Constructions. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 193, pp. 24:1–24:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.24, https://drops.dagstuhl.de/opus/volltexte/2021/13919

34. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theoretical computer science **192**(1), 3–29 (1998)
35. McBride, C.: Basics of bidirectionalism. https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/ (2018)
36. McBride, C.: Types who say ni. https://github.com/pigworker/TypesWhoSayNi/ (2022)
37. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic (TOCL) **9**(3), 1–49 (2008)
38. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
39. van Oostrom, V.: Normalisation in weakly orthogonal rewriting. In: International Conference on Rewriting Techniques and Applications. pp. 60–74. Springer (1999)
40. Pfenning, F.: Logical frameworks. Handbook of automated reasoning **2**, 1063–1147 (2001)
41. Pierce, B.C., Turner, D.N.: Local type inference. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(1), 1–44 (2000)
42. van Raamsdonk, F.: Outermost-fair rewriting. In: International Conference on Typed Lambda Calculi and Applications. pp. 284–299. Springer (1997)
43. Reed, J.: Redundancy elimination for lf. Electronic Notes in Theoretical Computer Science **199**, 89–106 (2008)
44. Sterling, J.: Algebraic type theory and universe hierarchies. arXiv preprint arXiv:1902.08848 (2019)
45. Uemura, T.: Abstract and concrete type theories. Ph.D. thesis, University of Amsterdam (2021)

Artifact report: Generic bidirectional typing for dependent type theories

Thiago Felicissimo(✉)

Université Paris-Saclay, INRIA, LMF, ENS Paris-Saclay, Gif-sur-Yvette, France
thiago.felicissimo@inria.fr

Abstract. We report on the implementation of a generic bidirectional algorithm for dependent type theories, following the proposal of the paper "Generic bidirectional typing for dependent type theories".

In [5] we have proposed a general definition of dependent type theories supporting bidirectional typing, and established an equivalence between their declarative and bidirectional type systems. The crucial property satisfied by the bidirectional system is its decidability for normalizing theories, which allowed for its implementation in OCaml in the tool `BiTTs` [6] which we describe here.

1 A quick introduction to the implementation

Let us first start with a concrete example of how to use the tool. Because the algorithm implemented is theory-independent, the first step to use it is to specify the theory we want to work in. This is done with the commands `sort`, `constructor`, `destructor` and `rewrite` which specify respectively sort, constructor, destructor and rewrite rules. For instance, the following declarations define the theory $\mathbb{T}_{\lambda\Pi}$ given in [5, Example 6], constituting a minimalistic Martin-Löf Type Theory with dependent functions.

```
sort Ty ()
sort Tm (A : Ty)
constructor Π () (A : Ty, B{x : Tm(A)} : Ty) : Ty
constructor λ (A : Ty, B{x : Tm(A)} : Ty) (t{x : Tm(A)} : Tm(B{x})) : Tm(Π(A, x. B{x}))
destructor @ (A : Ty, B{x : Tm(A)} : Ty) (t : Tm(Π(A, x. B{x}))) (u : Tm(A)) : Tm(B{u})
rewrite @(λ(x. t{x}), u) --> t{u}
```

Once the theory is specified, we can start writing and typechecking terms inside it. For instance, supposing we have also added a Tarski-style universe `U`, we can check the following definition of the polymorphic identity function.

```
let idU : Tm(Π(U, a. Π(El(a), _. El(a)))) := λ(a. λ(x. x))
```

To typecheck this definition, the tool first verifies that the sort given in the annotation is indeed well-typed, and then checks the body of the definition

© The Author(s) 2024
S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 171–175, 2024.
https://doi.org/10.1007/978-3-031-57262-3_7

against the sort. If all the steps succeed, the identifier is added to a global scope of top-level definitions and becomes available to be used in the rest of the file.

Supposing that the underlying theory is valid, Theorem 2 of [5] ensures that, if the implementation says that a term is well-typed, then the term is indeed well-typed in the declarative type system of the theory. Note however that the implementation does not currently check if the supplied theory is valid. Extending the implementation to check this automatically is future work, so for the time being this verification is left to the user.

Finally, we also provide commands for evaluating terms to normal form and checking that two terms are definitionally equal. For instance, assuming we have added natural numbers to the theory and defined factorial, we can use these commands to compute the factorial of 3 and check that it is equal to 6.

```
let fact3 : Tm(ℕ) := @(fact, S(S(S(0))))
eval fact3

let 6 : Tm(ℕ) := S(S(S(S(S(S(0))))))
check fact3 = 6
```

The implementation also comes with some examples of theories that can be defined in the framework, along with some examples of terms written in these theories. In the directory `examples/` we can find the following files:

- `mltt.bitt` : Martin-Löf Type Theory with a type-in-type Tarski-style universe, Π and Σ types, lists, booleans, and the unit, empty and W types.
- `mltt-coquand.bitt` : Martin-Löf Type Theory with a hierarchy of (weak) cumulative Coquand-style universes and universe polymorphism, with Π types and natural numbers.
- `hol.bitt` : Higher-Order Logic (also known as Simple Type Theory) with implication and universal quantification.

2 The implementation

The core of the implementation can be separated into two main parts: the typechecking and the normalization algorithms. Let us now discuss them in detail.

Normalization

Because the theories we support are dependently-typed, typechecking terms requires equality checking, which in turn requires reducing terms to normal form. In order to do so, we have implemented an untyped variant of *Normalization by Evaluation (NbE)*, inspired by the works of Coquand [4], Abel [1] and Kovacs [8]. In NbE, terms are evaluated into a separate syntax of runtime values, in which binders are represented by closures and free variables by unknowns. Values can then be compared for equality by entering closures and recursively evaluating and comparing their bodies. One of the benefits of this approach is that, by using de Bruijn indices in the syntax of regular terms but de Bruijn *levels* in the syntax of values, we completely avoid the need of implementing substitution or index-shifting functions.

Let us go through the main functions used to implement normalization. In the following, we only discuss those that operate on terms, but each one has a counterpart for metavariable substitutions. First, because the definitional equality of theories is generated by customizable rewrite systems, rewriting requires matching against patterns. This is done by the function

```
val match_tm : p_tm -> v_tm -> v_msubst
```

which matches a term value against a term pattern and produces a metavariable substitution of values (the prefix `p_` stands for pattern, while `v_` stands for value).

This is then used in the function

```
val eval_tm : tm -> v_msubst -> v_subst -> v_tm
```

which evaluates a term under a `v_subst` mapping occurring variables to values and a `v_msubst` mapping occurring metavariables to values or closures. This is done by recursively evaluating subterms, then trying to match against one of the rewrite rule's left hand sides and finally recursively evaluating the right hand side under the metavariable substitution returned by matching.

Finally, values can be checked for equality with the function

```
val equal_tm : v_tm -> v_tm -> int -> unit
```

which recursively enters and evaluates closures while checking the result for equality, and raising an exception when the two given terms are not equal. The third argument is used for generating fresh unknowns when entering closures.

Typechecking

The typechecking algorithm is composed of four main functions, each one implementing one of the judgment forms of the bidirectional system of [5].

Inference $\Gamma \vdash t \Rightarrow T$ and checking $\Gamma \vdash t \Leftarrow T$ are implemented by the functions

```
val infer : v_ctx -> v_subst -> tm -> v_tm
val check : v_ctx -> v_subst -> tm -> v_tm -> unit
```

the first returning the inferred sort and the second returning unit on success. Note that, following works such as [4,8,7], we tightly integrate it with the NbE algorithm by asking all inputs to be already in the syntax of values, with the exception of the subject of the typing judgment. Compared with the usual inference and checking judgments, note also the addition of the second argument `v_subst` used to map the variables of the context `v_ctx` to unknowns for when needing to evaluate the subject.

The third judgment $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{u} \Leftarrow \Xi$ used to typecheck metavariable substitutions is then rendered as the function

```
val check_msubst : v_ctx -> v_subst -> v_msubst -> msubst -> mctx -> v_msubst
```

in which the argument corresponding to Θ is omitted for it is computationally irrelevant. We also return the value of the checked metavariable substitution, which comes in handy when coding the recursive case of its definition. Finally, the last judgment $\Gamma \vdash T \Leftarrow \mathsf{sort}$ is implemented by the function

```
val check_sort : v_ctx -> v_subst -> tm -> unit
```

whose type signature follows the same reasoning as above.

Differences with respect to the theory

We highlight some relevant differences regarding the theory presented in [5].

First, we do not support matching inside binders, which restricts the set of patterns we can write. For instance, while the pattern `λ(x.t{x})` is accepted, the pattern `λ(x.S(t{x}))` (assuming that `S` is a constructor) would be rejected by the implementation. This is because matching against it would require matching inside a closure and then reading back the result into the syntax of terms, which would be highly inefficient with our NbE setup. Thankfully, matching inside binders is almost never needed and none of our provided examples require it.

Second, even though the inference system for matching and the proof of decidability of conversion in [5] employ the maximal outermost strategy, our NbE normalizer uses instead a call-by-value strategy. The maximal outermost strategy has the theoretical advantage over call-by-value of being normalizing, which means that it always terminates for weak normalizing theories. However, most theories used in practice are either strong normalizing or not normalizing at all. Moreover, call-by-value can be implemented very easily using our described NbE setup, which is the reason we opted for it instead.

Third, instead of defining the typing functions over a specific grammar of checkable/inferable terms as done in [5], we define them over the grammar of (regular) terms. This means that these functions might discover in the process that the term given is not checkable/inferable, in which case an error is given.

Finally, as seen in Section 1 our implementation also extends the bidirectional system with top-level definitions, which is crucial for allowing to write terms in a user-friendly manner.

3 Future work

The current implementation is still a prototype and can be extended in various ways. In particular, error handling is still rudimentary and improving it will be key in order to make `BiTTs` more user-friendly.

We also plan to further test our implementation with larger and more realistic examples. In particular, we would like to compare it with typecheckers for Dedukti [2,3], a framework aimed at providing a universal typechecker geared towards proof-system interoperability. Because Dedukti has no support for erased arguments, its terms are highly-annotated, which can have an important impact on performance. Our support for non-annotated syntaxes should therefore allow for shorter typechecking times, an hypothesis we hope to confirm with these tests.

References

1. Abel, A.: Normalization by evaluation: Dependent types and impredicativity. Habilitation. Ludwig-Maximilians-Universität München (2013)
2. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory (2016), unpublished
3. Blanqui, F., Dowek, G., Grienenberger, É., Hondet, G., Thiré, F.: Some axioms for mathematics. In: Kobayashi, N. (ed.) 6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference). LIPIcs, vol. 195, pp. 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.FSCD.2021.20, https://doi.org/10.4230/LIPIcs.FSCD.2021.20
4. Coquand, T.: An algorithm for type-checking dependent types. Science of Computer Programming **26**(1-3), 167–177 (1996)
5. Felicissimo, T.: Generic bidirectional typing for dependent type theories (2023)
6. Felicissimo, T.: BiTTs (Jan 2024). https://doi.org/10.5281/zenodo.10500598, https://doi.org/10.5281/zenodo.10500598
7. Gratzer, D., Sterling, J., Birkedal, L.: Implementing a modal dependent type theory. Proceedings of the ACM on Programming Languages **3**(ICFP), 1–29 (2019)
8. Kovács, A.: elaboration-zoo (2023), https://github.com/AndrasKovacs/elaboration-zoo

Deciding Subtyping for Asynchronous Multiparty Sessions

Elaine Li[1(✉)], Felix Stutz[2], and Thomas Wies[1]

[1] New York University, New York, USA
efl9013@nyu.edu, wies@cs.nyu.edu
[2] Max Planck Institute for Software Systems, Kaiserslautern, Germany
fstutz@mpi-sws.org

Abstract. Multiparty session types (MSTs) are a type-based approach to verifying communication protocols, represented as global types in the framework. We present a precise subtyping relation for asynchronous MSTs with communicating state machines (CSMs) as implementation model. We address two problems: when can a local implementation safely substitute another, and when does an arbitrary CSM implement a global type? We define safety with respect to a given global type, in terms of subprotocol fidelity and deadlock freedom. Our implementation model subsumes existing work which considers local types with restricted choice. We exploit the connection between MST subtyping and refinement to formulate concise conditions that are directly checkable on the candidate implementations, and use them to show that both problems are decidable in polynomial time.

Keywords: Protocol verification · Multiparty session types · Communicating state machines · Subtyping · Refinement.

1 Introduction

Multiparty session types (MSTs) [31] are a type-based approach to verifying communication protocols. In MST frameworks, a communication protocol is expressed as a *global type*, which describes the interactions of all protocol participants from a birds-eye view. The key property of interest in MST frameworks is *implementability*, which asks whether there exists a collection of local implementations, one per protocol participant, that is deadlock-free and produces the same set of behaviors described by the global type. The latter property is known as *protocol fidelity*. Given an implementable global type, the *synthesis* problem asks to compute such a collection. To solve implementability and synthesis, MST frameworks are often equipped with a *projection operator*, which is a partial map from global types to a collection of local implementations. Projection operators compute a correct implementation for a given global type if one exists.

However, projection operators only compute one candidate out of many possible implementations for a given global type, which narrows the usability of MST frameworks. As we demonstrate below, substituting this candidate can in some cases achieve an exponential reduction in the size of the local implementation. Furthermore, applications may sometimes require that an implementation produce only a subset of the

S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 176–205, 2024.
https://doi.org/10.1007/978-3-031-57262-3_8

global type's specified behaviors. We refer to this property as *subprotocol fidelity*. For example, a general client-server protocol may customize the set of requests it handles to the specific devices it runs on. Subtyping reintroduces this flexibility into MST frameworks, by characterizing when an implementation can replace another while preserving desirable correctness guarantees.

Formally, a subtyping relation is a reflexive and transitive relation that respects Liskov and Wing's substitution principle [39]: T' is a subtype of T when T' can be *safely* used in any context that expects a term of type T. While implementability for MSTs was originally defined on syntactic local types [29, 31], other implementation models have since been investigated, including communicating session automata [21] and behavioral contracts [16]. We motivate our work with the observation that a subtyping relation is only as powerful as its notion of safety, and the expressivity of its underlying implementation model. Existing subtyping relations adopt a notion of safety that is agnostic to a global specification. For example, [2, 3] define safety as the successful completion of a single role in binary sessions, [36] defines safety as eventual reception and progress of all roles in multiparty sessions, and [26] defines safety as the termination of all roles in multiparty sessions. As a result, these subtyping relations eagerly reject subtypes that are viable for the specific global type at hand. In addition, existing implementation models are restricted to local types with *directed choice* for branching, or equivalent representations thereof [9], which prohibit a role from sending messages to or receiving messages from different participants in a choice. This restrictiveness undermines the flexibility that subtyping is fundamentally designed to provide.

We present a subtyping relation that extends prior work along both dimensions. We define a stronger notion of safety with respect to a given global type: a substitution is safe if in all *well-behaved* contexts, the resulting implementation satisfies both deadlock freedom and subprotocol fidelity. We assume an implementation model of unrestricted communicating state machines (CSMs) [4] communicating via FIFO channels, which subsumes implementation models in prior work [20, 26, 36]. We demonstrate that this generalization renders existing subtyping relations which are precise for a restrictive implementation model incomplete. As a result of both extensions, our subtyping relation requires reasoning about available messages [40] for completeness, a novel feature that is absent from existing subtyping relations.

Our result applies to global types with *sender-driven* choice, which generalize global types from their original formulation with directed choice [31], and borrows insights from recent work on a sound and complete projection operator for this class of global types [38].

Contributions. In this paper, we present the first precise subtyping relation that guarantees deadlock freedom and subprotocol fidelity with respect to a global type, and that assumes an unrestricted, asynchronous CSM implementation model. We solve the *Protocol Verification* problem and the *Protocol Refinement* problem with respect to global type $\mathbf{G}$ and a set of roles $\mathcal{P}$:

1. *Protocol Verification*: Given a CSM $\mathcal{A}$, does $\mathcal{A}$ implement $\mathbf{G}$?
2. *Protocol Refinement*: Let $\mathtt{p}$ be a role and let B be a safe implementation for $\mathtt{p}$ in any well-behaved context for $\mathbf{G}$. Given A, can A safely replace B in any well-behaved context for $\mathbf{G}$?

We exploit the connection between MST subtyping and CSM refinement to formulate concise conditions that are directly checkable on candidate state machines. Using this characterization, we show that both problems are decidable in polynomial time.

2 Motivation

We first showcase that sound and complete projection operators can yield local implementations that are exponential in the size of its global type, but can be reduced to constant size by subtyping. We then demonstrate the restrictiveness of existing subtyping relations both in terms of their notion of safety and their implementation model.

Subset projection with exponentially many states. We first construct a family of implementable global types $\mathbf{G}_n$ for $n \in \mathbb{N}$ such that $\mathbf{G}_n$ has size linear in n and the deterministic finite state machine for $\mathtt{q}$ that recognizes the projection of the global language onto $\mathtt{q}$'s alphabet $\Sigma_{\mathtt{q}}$, denoted $\mathcal{L}(\mathbf{G}_n)\Downarrow_{\Sigma_{\mathtt{q}}}$, has size exponential in n.

The construction of the $\mathbf{G}_n$'s builds on the regular expression $(a^*(ab^*)^n a)^*$, which can only be recognized by a deterministic finite state machine that grows exponentially with n [23, Thm. 11].

First, we construct the part for $(ab^*)^i a$ recursively. In global types, $\mathtt{p}\!\to\!\mathtt{q}\!:\!m$ denotes role $\mathtt{p}$ sending a message m to role $\mathtt{q}$, $+$ denotes choice, μt binds a recursion variable t that can be used in the continuation, and 0 denotes termination.

$$G_i := \mathtt{p}\!\to\!\mathtt{q}\!:\!a.\,\mu t_{3,i}.\,+\begin{cases}\mathtt{p}\!\to\!\mathtt{r}\!:\!m_3.\,\mathtt{p}\!\to\!\mathtt{q}\!:\!b.\,t_{3,i}\\ \mathtt{p}\!\to\!\mathtt{r}\!:\!n_3.\,G_{i-1}\end{cases} \quad \text{for } i>0 \quad \text{and} \quad G_0 := \mathtt{p}\!\to\!\mathtt{q}\!:\!a.\,t_1$$

Here, each G_i for $i > 0$ generates (ab^*) and G_0 adds the last a. Role $\mathtt{p}$'s choice to send either m_3 or n_3 to $\mathtt{r}$ respectively encodes the choice to continue iterating b's or to stop in b^*; $\mathtt{q}$ however, is not involved in this exchange and thus $\mathtt{q}$'s local language is isomorphic to $(ab^*)^i a$.

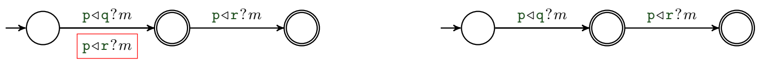
Next, we define some scaffolding $G(\text{-})$ for the outermost Kleene Star and the first a^*:

$$G(G') := \mu t_1.\,+\begin{cases}\mathtt{p}\!\to\!\mathtt{r}\!:\!m_1.\,\mu t_2.\,+\begin{cases}\mathtt{p}\!\to\!\mathtt{r}\!:\!m_2.\,\mathtt{p}\!\to\!\mathtt{q}\!:\!a.\,t_2\\ \mathtt{p}\!\to\!\mathtt{r}\!:\!n_2.\,G'\end{cases}\\ \mathtt{p}\!\to\!\mathtt{r}\!:\!n_1.\,0\end{cases}.$$

We combine both to obtain the family $\mathbf{G}_n := G(G_n)$.

As $\mathbf{G}_n$ is implementable, the subset projection [38] for each role is defined. One feature of the implementations computed by this projection operator is *local language preservation*, meaning that the language recognized by the local implementation is precisely the projection of the global language onto its alphabet, e.g. $\mathcal{L}(\mathbf{G}_n)\Downarrow_{\Sigma_{\mathtt{q}}}$ for role $\mathtt{q}$ with alphabet $\Sigma_{\mathtt{q}}$. In this case, because $\mathcal{L}(\mathbf{G}_n)\Downarrow_{\Sigma_{\mathtt{q}}}$ can only be recognized by a deterministic finite state machine with size exponential in n, the corresponding local language preserving implementation also has size exponential in n.

However, not all implementations need to satisfy local language preservation. Consider the type $\mu t.(\mathtt{p}\!\to\!\mathtt{q}\!:\!o.\,t + \mathtt{p}\!\to\!\mathtt{q}\!:\!b.\,0)$. The projection of the global language onto $\mathtt{q}$ limits $\mathtt{q}$ to only receiving a sequence of o messages terminated by a b message. However, an implementation for $\mathtt{q}$ can rely on $\mathtt{p}$ to send correct sequences of messages, and



(a) A (b) B

Fig. 1: Two state machines for role q

instead accept any message that it receives. A similar pattern arises in the family $\mathbf{G}_n$, where the exponentially-sized implementation for role q can simply be substituted with an automaton that allows to receive any message from p.

The restrictiveness of existing MST subtyping relations. Consider the two implementations for role p, represented as finite state machines A and B in Figs. 1a and 1b. State machine A embodies the idea of input covariance [25] by adding receive actions, namely $\boxed{\texttt{p}\triangleleft\texttt{q}?m}$, which denotes role p receiving a message m from role q. But is it the case that A is a subtype of B? A preliminary answer based on prior work [26, 34] is *no*, for the reason that A falls outside of the implementation models considered in these works: the initial state in A contains outgoing receive transitions from two distinct senders, q and r, and one of the final states contains an outgoing transition. Thus, there exists no local type representation of A.

As a first step, let us generalize the implementation model to machines with arbitrary finite state control, and revisit the question. It turns out that the answer now depends on what protocol role p, alongside the other roles in the context, is following. Consider the two global types

$$\mathbf{G}_1 := \texttt{q}\rightarrow\texttt{p}:m.\,\texttt{r}\rightarrow\texttt{p}:m.\,0 \qquad \text{and} \qquad \mathbf{G}_2 := \texttt{q}\rightarrow\texttt{p}:m.\,0 \quad .$$

We observe that A is a subtype of B under the context of $\mathbf{G}_2$, but not under the context of $\mathbf{G}_1$. Suppose that roles q and r are both following $\mathbf{G}_1$, and thus both roles send a message m to p. Under asynchrony, the two messages can arrive in p's channel in any order; this holds even in a synchronous setting. Therefore, there exists an execution trace in which p takes the transition labeled $\boxed{\texttt{p}\triangleleft\texttt{r}?m}$ in A and first receives from r. Role p then finds itself in a final state with a pending message from q that it is unable to receive, thus causing a deadlock in the CSM. On the other hand, if q were following $\mathbf{G}_2$, the addition of the receive transition $\boxed{\texttt{p}\triangleleft\texttt{r}?m}$ is safe because it is never enabled, and thus A can safely compose with any context following $\mathbf{G}_2$ without violating protocol fidelity and deadlock freedom.

3 Preliminaries

We restate relevant definitions from [38].

Words. Let Σ be a finite alphabet. Σ^* denotes the set of finite words over Σ, Σ^ω the set of infinite words, and Σ^∞ their union $\Sigma^* \cup \Sigma^\omega$. A word $u \in \Sigma^*$ is a *prefix* of word $v \in \Sigma^\infty$, denoted $u \leq v$, if there exists $w \in \Sigma^\infty$ with $u \cdot w = v$.

Message Alphabet. Let $\mathcal{P}$ be a set of roles and $\mathcal{V}$ be a set of messages. We define the set of *synchronous events* $\Sigma_{sync} := \{\mathtt{p}\to\mathtt{q}:m \mid \mathtt{p},\mathtt{q} \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$ where $\mathtt{p}\to\mathtt{q}:m$ denotes that message m is sent by $\mathtt{p}$ to $\mathtt{q}$ atomically. This is split for *asynchronous events*. For a role $\mathtt{p} \in \mathcal{P}$, we define the alphabet $\Sigma_{\mathtt{p},!} = \{\mathtt{p}\triangleright\mathtt{q}!m \mid \mathtt{q} \in \mathcal{P},\ m \in \mathcal{V}\}$ of *send* events and the alphabet $\Sigma_{\mathtt{p},?} = \{\mathtt{p}\triangleleft\mathtt{q}?m \mid \mathtt{q} \in \mathcal{P},\ m \in \mathcal{V}\}$ of *receive* events. The event $\mathtt{p}\triangleright\mathtt{q}!m$ denotes role $\mathtt{p}$ sending a message m to $\mathtt{q}$, and $\mathtt{p}\triangleleft\mathtt{q}?m$ denotes role $\mathtt{p}$ receiving a message m from $\mathtt{q}$. We write $\Sigma_{\mathtt{p}} = \Sigma_{\mathtt{p},!} \cup \Sigma_{\mathtt{p},?}$, $\Sigma_! = \bigcup_{\mathtt{p}\in\mathcal{P}} \Sigma_{\mathtt{p},!}$, and $\Sigma_? = \bigcup_{\mathtt{p}\in\mathcal{P}} \Sigma_{\mathtt{p},?}$. Finally, $\Sigma_{async} = \Sigma_! \cup \Sigma_?$. We say that $\mathtt{p}$ is *active* in $x \in \Sigma_{async}$ if $x \in \Sigma_{\mathtt{p}}$. For each role $\mathtt{p} \in \mathcal{P}$, we define a homomorphism $\Downarrow_{\Sigma_{\mathtt{p}}}$, where $x\Downarrow_{\Sigma_{\mathtt{p}}} = x$ if $x \in \Sigma_{\mathtt{p}}$ and ε otherwise. We fix $\mathcal{P}$ and $\mathcal{V}$ in the rest of the paper.

Global Types – Syntax. Global types for MSTs [40] are defined by the grammar:

$$G ::= 0 \ \mid \ \sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i \ \mid \ \mu t.\, G \ \mid \ t$$

where $\mathtt{p}, \mathtt{q}_i$ range over $\mathcal{P}$, m_i over $\mathcal{V}$, and t over a set of recursion variables.

We require each branch of a choice to be distinct: $\forall i,j \in I.\, i \neq j \Rightarrow (\mathtt{q}_i, m_i) \neq (\mathtt{q}_j, m_j)$, the sender and receiver of an event to be distinct: $\mathtt{p} \neq \mathtt{q}_i$ for each $i \in I$, and recursion to be guarded: in $\mu t.\, G$, there is at least one message between μt and each t in G. We omit $\sum$ for singleton choices. When working with a protocol described by a global type, we use $\mathbf{G}$ to refer to the top-level type, and G to refer to its subterms.

We use the extended definition of global types from [40] featuring *sender-driven choice*. This definition subsumes classical MSTs that only allow *directed choice* [31]. We focus on communication primitives and omit features like delegation or parametrization, and refer the reader to §7 for a discussion of different MST frameworks.

Global Types – Semantics. As a basis for the semantics of a global type $\mathbf{G}$, we construct a finite state machine $\mathsf{GAut}(\mathbf{G}) = (Q_{\mathbf{G}}, \Sigma_{sync}, \delta_{\mathbf{G}}, q_{0,\mathbf{G}}, F_{\mathbf{G}})$ where

– $Q_{\mathbf{G}}$ is the set of all syntactic subterms in $\mathbf{G}$ together with the term 0,
– $\delta_{\mathbf{G}}$ consists of the transitions $(\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i, \mathtt{p}\to\mathtt{q}_i : m_i, G_i)$ for each $i \in I$, as well as $(\mu t.G', \varepsilon, G')$ and $(t, \varepsilon, \mu t.G')$ for each subterm $\mu t.G'$,
– $q_{0,\mathbf{G}} = \mathbf{G}$ and $F_{\mathbf{G}} = \{0\}$.

We define a homomorphism `split` onto the asynchronous alphabet:

$$\mathtt{split}(\mathtt{p}\to\mathtt{q}:m) := \mathtt{p}\triangleright\mathtt{q}!m.\,\mathtt{q}\triangleleft\mathtt{p}?m \ .$$

The semantics $\mathcal{L}(\mathbf{G})$ of a global type $\mathbf{G}$ is given by $\mathcal{C}^{\sim}(\mathtt{split}(\mathcal{L}(\mathsf{GAut}(\mathbf{G}))))$ where $\mathcal{C}^{\sim}$ is the closure under the indistinguishability relation $\sim$ [40]. Two events are independent if they are not related by the *happened-before* relation [33]. For instance, any two send events from distinct senders are independent. Two words are indistinguishable if one can be reordered into the other by repeatedly swapping consecutive independent events. The full definition can be found in the extended version [37].

We call a state $q_G \in Q_{\mathbf{G}}$ a *send-originating* state, denoted $q_G \in Q_{\mathbf{G},!}$ for role $\mathtt{p}$ if there exists a transition $q_G \xrightarrow{\mathtt{p}\to\mathtt{q}:m} q_{G'} \in \delta_{\mathbf{G}}$, and a *receive-originating* state, denoted $q_G \in Q_{\mathbf{G},?}$ for $\mathtt{p}$ if there exists a transition $q_G \xrightarrow{\mathtt{q}\to\mathtt{p}:m} q_{G'} \in \delta_{\mathbf{G}}$. We omit mention of role $\mathtt{p}$ when clear from context.

Communicating State Machine [4]. $\mathcal{A} = \{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is a CSM over $\mathcal{P}$ and $\mathcal{V}$ if $A_{\mathtt{p}} = (Q_{\mathtt{p}}, \Sigma_{\mathtt{p}}, \delta_{\mathtt{p}}, q_{0,\mathtt{p}}, F_{\mathtt{p}})$ is a deterministic finite state machine over $\Sigma_{\mathtt{p}}$ for every $\mathtt{p} \in \mathcal{P}$. Let $\prod_{\mathtt{p}\in\mathcal{P}} Q_{\mathtt{p}}$ denote the set of global states and $\mathsf{Chan} = \{(\mathtt{p},\mathtt{q}) \mid \mathtt{p},\mathtt{q} \in \mathcal{P}, \mathtt{p} \neq \mathtt{q}\}$ denote the set of channels. A *configuration* of $\mathcal{A}$ is a pair $(\vec{s}, \xi)$, where $\vec{s}$ is a global state and $\xi : \mathsf{Chan} \to \mathcal{V}^*$ is a mapping from each channel to a sequence of messages. We use $\vec{s}_{\mathtt{p}}$ to denote the state of $\mathtt{p}$ in $\vec{s}$. The CSM transition relation, denoted $\to$, is defined as follows.

- $(\vec{s}, \xi) \xrightarrow{\mathtt{p}\triangleright\mathtt{q}!m} (\vec{s}\,', \xi')$ if $(\vec{s}_{\mathtt{p}}, \mathtt{p}\triangleright\mathtt{q}!m, \vec{s}\,'_{\mathtt{p}}) \in \delta_{\mathtt{p}}$, $\vec{s}_{\mathtt{r}} = \vec{s}\,'_{\mathtt{r}}$ for every role $\mathtt{r} \neq \mathtt{p}$, $\xi'(\mathtt{p},\mathtt{q}) = \xi(\mathtt{p},\mathtt{q}) \cdot m$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \mathsf{Chan}$.
- $(\vec{s}, \xi) \xrightarrow{\mathtt{q}\triangleleft\mathtt{p}?m} (\vec{s}\,', \xi')$ if $(\vec{s}_{\mathtt{q}}, \mathtt{q}\triangleleft\mathtt{p}?m, \vec{s}\,'_{\mathtt{q}}) \in \delta_{\mathtt{q}}$, $\vec{s}_{\mathtt{r}} = \vec{s}\,'_{\mathtt{r}}$ for every role $\mathtt{r} \neq \mathtt{q}$, $\xi(\mathtt{p},\mathtt{q}) = m \cdot \xi'(\mathtt{p},\mathtt{q})$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \mathsf{Chan}$.

In the initial configuration $(\vec{s}_0, \xi_0)$, each role's state in $\vec{s}_0$ is the initial state $q_{0,\mathtt{p}}$ of $A_{\mathtt{p}}$, and ξ_0 maps each channel to ε. A configuration $(\vec{s}, \xi)$ is said to be *final* iff $\vec{s}_{\mathtt{p}}$ is final for every $\mathtt{p}$ and ξ maps each channel to ε. Runs and traces are defined in the expected way. A run is *maximal* if either it is finite and ends in a final configuration, or it is infinite. The language $\mathcal{L}(\mathcal{A})$ of the CSM $\mathcal{A}$ is defined as the set of maximal traces. A configuration $(\vec{s}, \xi)$ is a *deadlock* if it is not final and has no outgoing transitions. A CSM is *deadlock-free* if no reachable configuration is a deadlock.

Definition 3.1 (Implementability). *We say that a CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ implements a global type $\mathbf{G}$ if the following two properties hold: (i)* protocol fidelity: $\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}(\mathbf{G})$*, and (ii)* deadlock freedom: $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *is deadlock-free. A global type $\mathbf{G}$ is* implementable *if there exists a CSM that implements it.*

One candidate implementation for global types can be computed directly from $\mathsf{GAut}(\mathbf{G})$, by removing actions unrelated to each role and determinizing the result. The following two definitions define this candidate implementation in two steps.

Definition 3.2 (Projection by Erasure [38]). *Let $\mathbf{G}$ be some global type with its state machine $\mathsf{GAut}(\mathbf{G}) = (Q_{\mathbf{G}}, \Sigma_{sync}, \delta_{\mathbf{G}}, q_{0,\mathbf{G}}, F_{\mathbf{G}})$. For each role $\mathtt{p} \in \mathcal{P}$, we define the state machine $\mathsf{GAut}(\mathbf{G})\!\downarrow_{\mathtt{p}} = (Q_{\mathbf{G}}, \Sigma_{\mathtt{p}} \uplus \{\varepsilon\}, \delta_{\downarrow}, q_{0,\mathbf{G}}, F_{\mathbf{G}})$ where $\delta_{\downarrow} := \{q \xrightarrow{\mathtt{split}(a)\Downarrow_{\Sigma_{\mathtt{p}}}} q' \mid q \xrightarrow{a} q' \in \delta_{\mathbf{G}}\}$. By definition of* $\mathtt{split}(\text{-})$*, it holds that* $\mathtt{split}(a)\Downarrow_{\Sigma_{\mathtt{p}}} \in \Sigma_{\mathtt{p}} \uplus \{\varepsilon\}$.

We determinize $\mathsf{GAut}(\mathbf{G})\!\downarrow_{\mathtt{p}}$ via a standard subset construction [46] to obtain a deterministic local state machine for $\mathtt{p}$. Note that the construction ensures that $Q_{\mathtt{p}}$ only contains subsets of $Q_{\mathbf{G}}$ whose states are reachable via the same traces.

Definition 3.3 (Subset Construction [38]). *Let $\mathbf{G}$ be a global type and $\mathtt{p}$ be a role. Then, the* subset construction *for $\mathtt{p}$ is defined as*

$$\mathscr{C}(\mathbf{G}, \mathtt{p}) = \big(Q_{\mathtt{p}}, \Sigma_{\mathtt{p}}, \delta_{\mathtt{p}}, s_{0,\mathtt{p}}, F_{\mathtt{p}}\big) \textit{ where}$$

- $\delta(s, a) := \{q' \in Q_{\mathbf{G}} \mid \exists q \in s, q \xrightarrow{a}\xrightarrow{\varepsilon}{}^{*} q' \in \delta_{\downarrow}\}$*, for every $s \subseteq Q_{\mathbf{G}}$ and $a \in \Sigma_{\mathtt{p}}$,*
- $s_{0,\mathtt{p}} := \{q \in Q_{\mathbf{G}} \mid q_{0,\mathbf{G}} \xrightarrow{\varepsilon}{}^{*} q \in \delta_{\downarrow}\}$,
- $Q_{\mathtt{p}} := \mathrm{lfp}^{\subseteq}_{\{s_{0,\mathtt{p}}\}} \lambda Q.\, Q \cup \{\delta(s, a) \mid s \in Q \wedge a \in \Sigma_{\mathtt{p}}\} \setminus \{\emptyset\}$,

- $\delta_{\mathtt{p}} := \delta|_{Q_{\mathtt{p}} \times \Sigma_{\mathtt{p}}}$, *and*
- $F_{\mathtt{p}} := \{s \in Q_{\mathtt{p}} \mid s \cap F_{\mathbf{G}} \neq \emptyset\}$.

Li et al. [38] showed that if $\mathbf{G}$ is implementable, then $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ implements $\mathbf{G}$ and satisfies the following property:

Definition 3.4. *Let* $\mathbf{G}$ *be a global type. We call an implementation* $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ local language preserving *with respect to* $\mathbf{G}$ *if* $\mathcal{L}(A_{\mathtt{p}}) = \mathcal{L}(\mathbf{G})\Downarrow_{\Sigma_{\mathtt{p}}}$ *for all* $\mathtt{p} \in \mathcal{P}$.

For the remainder of the paper, we fix a global type $\mathbf{G}$ that we assume is implementable.

4 Deciding Protocol Verification

Protocol Verification asks: Given a CSM $\mathcal{A}$, does $\mathcal{A}$ implement $\mathbf{G}$? For two CSMs $\mathcal{A}$ and $\mathcal{B}$, we say that $\mathcal{A}$ refines $\mathcal{B}$ if and only if every trace in $\mathcal{A}$ is a trace in $\mathcal{B}$, and a trace in $\mathcal{A}$ terminates maximally in $\mathcal{A}$ if and only if it terminates maximally in $\mathcal{B}$. If $\mathcal{A}$ and $\mathcal{B}$ refine each other, we say that they are equivalent. Further, in the case that $\mathcal{B}$ is deadlock-free, one can simplify the condition to the following: every trace in $\mathcal{A}$ is a trace in $\mathcal{B}$, and if a trace terminates in $\mathcal{A}$, then it terminates in $\mathcal{B}$ and is maximal in $\mathcal{A}$.

We can recast *Protocol Verification* in terms of CSM refinement using the fact that $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ is an implementation for $\mathbf{G}$. Therefore, the question amounts to asking whether $\mathcal{A}$ and $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ are equivalent.

Our goal is then to present a characterization C_1 that satisfies the following:

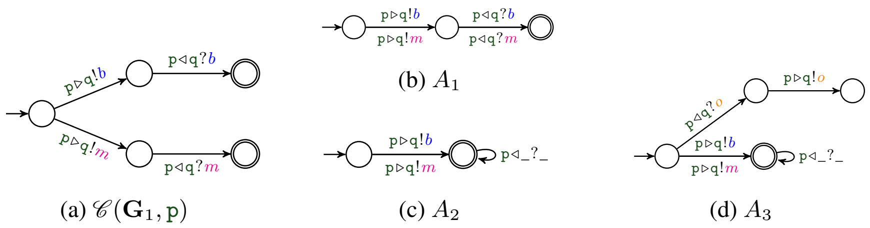
Theorem 4.1. *Let* $\mathbf{G}$ *be an implementable global type and* $\mathcal{A}$ *be a CSM. Then, the subset construction* $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ *and* $\mathcal{A}$ *are equivalent if and only if* C_1 *is satisfied.*

We motivate our characterization for *Protocol Verification* using a series of examples. Consider the following simple global type $\mathbf{G}_1$:

$$\mathbf{G}_1 := + \begin{cases} \mathtt{p} \to \mathtt{q} : b.\ \mathtt{q} \to \mathtt{p} : b.\ 0 \\ \mathtt{p} \to \mathtt{q} : m.\ \mathtt{q} \to \mathtt{p} : m.\ 0 \end{cases}$$

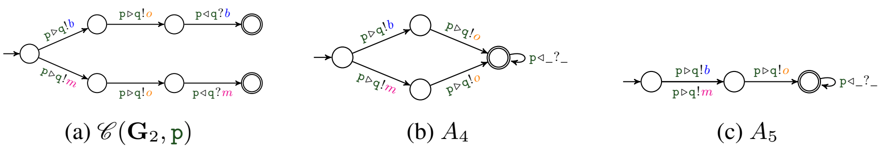
This global type is trivially implementable; the subset construction for role $\mathtt{q}$ obtained by the projection operator in [38] is depicted in Fig. 2a. Clearly, in any CSM implementing $\mathbf{G}_1$, the subset construction can be replaced with the more compact state machine A_1, shown in Fig. 2b.

For a local state machine in a CSM, control flow is determined by both the local transition relation and the global channel state. However, in some cases, the local information is redundant: the role's channel contents alone are enough to enforce that it produces the correct behaviors. In the example above, after $\mathtt{p}$ chooses to send $\mathtt{q}$ either m or b, $\mathtt{q}$ will guarantee that the correct message, i.e. the same one, is sent back to $\mathtt{p}$. Role $\mathtt{p}$'s state machine can rely on its channel contents to follow the protocol – it does not need separate control states for each message. In fact, we can further replace $\mathtt{p}$'s control states after sending with an accepting universal receive state, as shown in A_2 in Fig. 2c. Finally, we can add send transitions from unreachable states, as shown in A_3 in Fig. 2d.



(a) $\mathscr{C}(\mathbf{G}_1, \mathtt{p})$ (b) A_1 (c) A_2 (d) A_3

Fig. 2: Subset construction of $\mathbf{G}_1$ onto $\mathtt{p}$ and three alternative implementations



(a) $\mathscr{C}(\mathbf{G}_2, \mathtt{p})$ (b) A_4 (c) A_5

Fig. 3: Subset construction of $\mathbf{G}_2$ onto $\mathtt{p}$ and two alternative implementations

Similar patterns arise for send actions. Consider the following variation of the first global type, $\mathbf{G}_2$:

$$\mathbf{G}_2 := + \begin{cases} \mathtt{p} \to \mathtt{q} : b. \mathtt{p} \to \mathtt{r} : o. \mathtt{q} \to \mathtt{p} : b. 0 \\ \mathtt{p} \to \mathtt{q} : m. \mathtt{p} \to \mathtt{r} : o. \mathtt{q} \to \mathtt{p} : m. 0 \end{cases}$$

The subset construction from [38] yields the state machine for $\mathtt{p}$ shown in Fig. 3a.

Our reasoning above shows that A_4, depicted in Fig. 3b, is a correct alternative implementation for $\mathtt{p}$. Now observe that the pre-states of the two $\mathtt{p}\triangleright\mathtt{q}!o$ transitions can be collapsed because their continuations are identical. This yields another correct alternative implementation A_5, shown in Fig. 3c.

Informally, the subset construction takes a "maximalist" approach, creating as many distinct states as possible from the global type, and checking whether they are enough to guarantee that the role behaves correctly. However, sometimes this maximalism creates redundancy: just because two states are distinct according to the global type does not mean they need to be. In these cases, an implementation has the flexibility to merge certain distinct states together, or add transitions to a state. We wish to precisely characterize when such modifications to local state machines preserve protocol fidelity and deadlock freedom.

Our conditions for C_1 are inspired by the Send and Receive Validity conditions that precisely characterize implementability for global types, given in [38]. We restate the conditions, in addition to relevant definitions, for clarity.

Definition 4.2 (Available messages [40]). *The set of available messages is recursively defined on the structure of the global type. For completeness, we need to unfold the distinct recursion variables once. For this, we define a map $get\mu$ from variable to subterms and write $get\mu_{\mathbf{G}}$ for $get\mu(\mathbf{G})$:*

$$get\mu(0) := [\,] \qquad get\mu(t) := [\,] \qquad get\mu(\mu t.G) := [t \mapsto G] \cup get\mu(G)$$
$$get\mu(\textstyle\sum_{i\in I} \mathtt{p} \to \mathtt{q}_i : m_i.G_i) := \bigcup_{i\in I} get\mu(G_i)$$

The function $M^{\mathcal{B},T}_{(\text{-}\ldots)}$ keeps a set of unfolded variables T, which is empty initially.

$$M^{\mathcal{B},T}_{(0\ldots)} := \emptyset \qquad M^{\mathcal{B},T}_{(\mu t.G\ldots)} := M^{\mathcal{B},T\cup\{t\}}_{(G\ldots)} \qquad M^{\mathcal{B},T}_{(t\ldots)} := \begin{cases} \emptyset & \textit{if } t \in T \\ M^{\mathcal{B},T\cup\{t\}}_{(get\mu_{\mathbf{G}}(t)\ldots)} & \textit{if } t \notin T \end{cases}$$

$$M^{\mathcal{B},T}_{(\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i\ldots)} := \begin{cases} \bigcup_{i\in I, m\in\mathcal{V}}(M^{\mathcal{B},T}_{(G_i\ldots)} \setminus \{\mathtt{p}\triangleright\mathtt{q}_i!m\}) \cup \{\mathtt{p}\triangleright\mathtt{q}_i!m_i\} & \textit{if } \mathtt{p} \notin \mathcal{B} \\ \bigcup_{i\in I} M^{\mathcal{B}\cup\{\mathtt{q}_i\},T}_{(G_i\ldots)} & \textit{if } \mathtt{p} \in \mathcal{B} \end{cases}$$

We write $M^{\mathcal{B}}_{(G\ldots)}$ *for* $M^{\mathcal{B},\emptyset}_{(G\ldots)}$. *If* $\mathcal{B}$ *is a singleton set, we omit set notation and write* $M^{\mathtt{p}}_{(G\ldots)}$ *for* $M^{\{\mathtt{p}\}}_{(G\ldots)}$.

Intuitively, the available messages definition captures all of the messages that can be at the head of their respective channels when a particular role is blocked from taking further transitions.

For notational convenience, we define the *origin* and *destination* of a transition following [38], but generalized from the subset construction automaton.

Definition 4.3 (Transition Origin and Destination). *Let* $\mathbf{G}$ *be a global type and let* $\delta_\downarrow$ *be the transition relation of* $\mathsf{GAut}(\mathbf{G})\Downarrow_{\mathtt{p}}$. *For* $x \in \Sigma_{\mathtt{p}}$ *and* $s, s' \subseteq Q_{\mathbf{G}}$, *we define the set of* transition origins $\text{tr-orig}(s \xrightarrow{x} s')$ *and* transition destinations $\text{tr-dest}(s \xrightarrow{x} s')$ *as follows:*

$$\text{tr-orig}(s \xrightarrow{x} s') := \{G \in s \mid \exists G' \in s'.\, G \xrightarrow{x}{}^{*} G' \in \delta_\downarrow\} \;\; \textit{and}$$
$$\text{tr-dest}(s \xrightarrow{x} s') := \{G' \in s' \mid \exists G \in s.\, G \xrightarrow{x}{}^{*} G' \in \delta_\downarrow\} \;\; .$$

Li et al. [38] showed that $\mathbf{G}$ is implementable if and only if the subset construction CSM $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ satisfies Send and Receive Validity for each $\mathscr{C}(\mathbf{G},\mathtt{p})$.

Definition 4.4 (Send Validity). $\mathscr{C}(\mathbf{G},\mathtt{p})$ *satisfies* Send Validity *iff every send transition* $s \xrightarrow{x} s' \in \delta_{\mathtt{p}}$ *is enabled in all states contained in* s*:*

$$\forall s \xrightarrow{x} s' \in \delta_{\mathtt{p}}.\; x \in \Sigma_{\mathtt{p},!} \implies \text{tr-orig}(s \xrightarrow{x} s') = s \;\; .$$

Definition 4.5 (Receive Validity). $\mathscr{C}(\mathbf{G},\mathtt{p})$ *satisfies* Receive Validity *iff no receive transition is enabled in an alternative continuation that originates from the same source state:*

$$\forall s \xrightarrow{\mathtt{p}\triangleleft\mathtt{q}_1?m_1} s_1,\; s \xrightarrow{\mathtt{p}\triangleleft\mathtt{q}_2?m_2} s_2 \in \delta_{\mathtt{p}}.$$
$$\mathtt{q}_1 \neq \mathtt{q}_2 \implies \forall\, G_2 \in \text{tr-dest}(s \xrightarrow{\mathtt{p}\triangleleft\mathtt{q}_2?m_2} s_2).\; \mathtt{q}_1\triangleright\mathtt{p}!m_1 \notin M^{\mathtt{p}}_{(G_2\ldots)} \;\; .$$

We wish to adapt these conditions to define C_1. However, unlike Send and Receive Validity, which are defined on special state machines, namely the subset construction for each role, the *Protocol Verification* problem asks whether arbitrary state machines implement the given $\mathbf{G}$.

We first present a *state decoration* function which maps local states in an arbitrary deterministic finite state machine to sets of global states in $\mathbf{G}$. Intuitively, state decoration captures all global states that can be reached in the projection by erasure automaton $\mathsf{GAut}(\mathbf{G})\Downarrow_{\mathtt{q}}$ on the same prefixes that reach the present state in the local state machine.

Definition 4.6 (State decoration with respect to G). *Let* $\mathsf{p} \in \mathcal{P}$ *be a role and let* $A = (Q, \Sigma_{\mathsf{p}}, s_0, \delta, F)$ *be a deterministic finite state machine for* p. *Let* $\mathsf{GAut}(\mathbf{G})\!\downarrow_{\mathsf{p}} = (Q_{\mathbf{G}}, \Sigma_{\mathsf{p}} \uplus \{\varepsilon\}, \delta_{\downarrow}, q_{0,\mathbf{G}}, F_{\mathbf{G}})$ *be* p*'s projection by erasure state machine for* $\mathbf{G}$. *We define a total function* $d_{\mathbf{G},A} : Q \to 2^{Q_{\mathbf{G}}}$ *that maps each state in* A *to a subset of states in* $\mathsf{GAut}(\mathbf{G})\!\downarrow_{\mathsf{p}}$ *such that:*

$$d_{\mathbf{G},A,\mathsf{p}}(s) = \{q \in Q_{\mathbf{G}} \mid \exists u \in \Sigma_{\mathsf{p}}^*.\ s_0 \xrightarrow{u}{}^* s \in \delta \land q_{0,\mathbf{G}} \xrightarrow{u}{}^* q \in \delta_{\downarrow}\}\ .$$

We refer to $d_{\mathbf{G},A,\mathsf{p}}(s)$ *as the* decoration set *of* s, *and omit the subscripts* $\mathbf{G}, A, \mathsf{p}$ *when clear from context.*

Remark 4.7. Note that the subset construction can be viewed as a special state machine for which the state decoration function is the identity function. In other words, for all $s \in Q_{\mathsf{p}}$ where Q_{p} is the set of states of $\mathscr{C}(\mathbf{G}, \mathsf{p})$, $d(s) = s$.

We are now equipped to present C_1.

Definition 4.8 **(C_1).** *Let* $\mathbf{G}$ *be a global type and* $\mathcal{A}$ *be a CSM.* C_1 *is satisfied when for all* $\mathsf{p} \in \mathcal{P}$, *with* $A_{\mathsf{p}} = (Q_{\mathsf{p}}, \Sigma_{\mathsf{p}}, \delta_{\mathsf{p}}, s_{0,\mathsf{p}}, F_{\mathsf{p}})$ *denoting the state machine for* p *in* $\mathcal{A}$, *the following conditions hold:*

- Send Decoration Validity*: every send transition* $s \xrightarrow{x} s' \in \delta_{\mathsf{p}}$ *is enabled in all states decorating* s*:*
  $\forall s \xrightarrow{\mathsf{p}\triangleright\mathsf{q}!m} s' \in \delta_{\mathsf{p}}.\ \text{tr-orig}(d(s) \xrightarrow{\mathsf{p}\triangleright\mathsf{q}!m} d(s')) = d(s)$.
- Receive Decoration Validity*: no receive transition is enabled in an alternative continuation originating from the same state:*
  $\forall s \xrightarrow{\mathsf{p}\triangleleft\mathsf{q}_1?m_1} s_1,\ s \xrightarrow{x} s_2 \in \delta_{\mathsf{p}}.\ x \neq \mathsf{p}\triangleleft\mathsf{q}_1?\_ \implies$
  $\forall G' \in \text{tr-dest}(d(s) \xrightarrow{x} d(s_2)).\ \mathsf{q}_1\triangleright\mathsf{p}!m_1 \notin M^{\mathsf{p}}_{(G'\ldots)}$.
- Transition Exhaustivity*: every transition that is enabled in some global state decorating* s *must be an outgoing transition from* s*:*
  $\forall s \in Q.\ \forall G \xrightarrow{x}{}^* G' \in \delta_{\downarrow}.\ G \in d(s) \implies \exists s' \in Q.\ s \xrightarrow{x} s' \in \delta_{\mathsf{p}}$.
- Final State Validity*: a reachable state with a non-empty decorating set is final if its decorating set contains a final global state:*
  $\forall s \in Q.\ d(s) \neq \emptyset \implies (d(s) \cap F_{\mathbf{G}} \neq \emptyset \implies s \in F_{\mathsf{p}})$.

We want to show the following equivalence to prove Theorem 4.1:

$$C_1 \Leftrightarrow \mathcal{A} \text{ refines } \{\!\{\mathscr{C}(\mathbf{G}, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}} \text{ and } \{\!\{\mathscr{C}(\mathbf{G}, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}} \text{ refines } \mathcal{A}.$$

We address soundness (the forward direction) and completeness (the backward direction) in turn. Soundness states that C_1 is sufficient to show that $\mathcal{A}$ preserves all behaviors of the subset construction, and does not introduce new behaviors.

We say that a state machine A for role p satisfies *Local Language Inclusion* if it satisfies $\mathcal{L}(\mathbf{G})\Downarrow_{\Sigma_{\mathsf{p}}} \subseteq \mathcal{L}(A)$. The following lemma, proven in the extended version [37], establishes that *Local Language Inclusion* follows from *Transition Exhaustivity* and *Final State Validity*.

Lemma 4.9. *Let $A_{\mathtt{p}} = (Q_{\mathtt{p}}, \Sigma_{\mathtt{p}}, \delta_{\mathtt{p}}, s_{0,\mathtt{p}}, F_{\mathtt{p}})$ denote the state machine for* p *in $\mathcal{A}$. Then,* Transition Exhaustivity *and* Final State Validity *imply $\mathcal{L}(\mathbf{G})\Downarrow_{\Sigma_{\mathtt{p}}} \subseteq \mathcal{L}(A_{\mathtt{p}})$.*

The fact that $\mathcal{A}$ preserves behaviors follows immediately from *Local Language Inclusion*. The fact that $\mathcal{A}$ does not introduce new behaviors, on the other hand, is enforced by *Send Decoration Validity* and *Receive Decoration Validity*.

In the soundness proof for each of our conditions, we prove refinement via structural induction on traces. We show refinement in two steps, first showing that any trace in one CSM is a trace in the other, and then showing that any terminated trace in one CSM is terminated in the other and maximal.

We recall two definitions from [38] used in the soundness proof.

Definition 4.10 (Intersection sets). *Let $\mathbf{G}$ be a global type and $\mathsf{GAut}(\mathbf{G})$ be the corresponding state machine. Let* p *be a role and $w \in \Sigma_{async}^*$ be a word. We define the set of possible runs $\mathrm{R}_{\mathtt{p}}^{\mathbf{G}}(w)$ as all maximal runs of $\mathsf{GAut}(\mathbf{G})$ that are consistent with* p*'s local view of w:*

$$\mathrm{R}_{\mathtt{p}}^{\mathbf{G}}(w) := \{\rho \textit{ is a maximal run of } \mathsf{GAut}(\mathbf{G}) \mid w\Downarrow_{\Sigma_{\mathtt{p}}} \leq \mathtt{split}(\mathtt{trace}(\rho))\Downarrow_{\Sigma_{\mathtt{p}}}\} \ .$$

We denote the intersection of the possible run sets for all roles as

$$I(w) := \bigcap_{\mathtt{p}\in\mathcal{P}} \mathrm{R}_{\mathtt{p}}^{\mathbf{G}}(w) \ .$$

Definition 4.11 (Unique splitting of a possible run). *Let $\mathbf{G}$ be a global type,* p *a role, and $w \in \Sigma_{async}^*$ a word. Let ρ be a possible run in $\mathrm{R}_{\mathtt{p}}^{\mathbf{G}}(w)$. We define the longest prefix of ρ matching w:*

$$\alpha' := \max\{\rho' \mid \rho' \leq \rho \ \wedge \ \mathtt{split}(\mathtt{trace}(\rho'))\Downarrow_{\Sigma_{\mathtt{p}}} \leq w\Downarrow_{\Sigma_{\mathtt{p}}}\} \ .$$

If $\alpha' \neq \rho$, we can split ρ into $\rho = \alpha \cdot G \xrightarrow{l} G' \cdot \beta$ where $\alpha' = \alpha \cdot G$, G' denotes the state following G, and β denotes the suffix of ρ following $\alpha \cdot G \cdot G'$. We call $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ the unique splitting of ρ for p *matching w. We omit the role* p *when obvious from context. This splitting is always unique because the maximal prefix of any $\rho \in \mathrm{R}_{\mathtt{p}}^{\mathbf{G}}(w)$ matching w is unique.*

Lemma 4.12 (Soundness of C_1). *C_1 implies that $\mathcal{A}$ and $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ are equivalent.*

Proof. The proof that C_1 implies $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ refines $\mathcal{A}$ depends only on *Local Language Inclusion* and can be straightforwardly adapted from [38, Lemma 4.4]. We instead focus on showing that C_1 implies $\mathcal{A}$ refines $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$, which depends on the other two conditions in C_1. First, we prove that any trace in $\mathcal{A}$ is a trace in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$:
Claim 1: $\forall\, w \in \Sigma_{async}^{\infty}$. w is a trace in $\mathcal{A}$ implies w is a trace in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$.

We prove the claim by induction for all finite w. The infinite case follows from the finite case because $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deterministic and all prefixes of w are traces of $\mathcal{A}$ and, hence, of $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$. The base cases, where $w = \varepsilon$, is trivially discharged by the fact that ε is a trace of all CSMs. In the inductive step, assume that w is a trace of $\mathcal{A}$. Let $x \in \Sigma_{async}$ such that wx is a trace of $\mathcal{A}$. We want to show that wx is also a trace of $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$.

From the induction hypothesis, we know that w is a trace of $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$. Let ξ be the channel configuration uniquely determined by w. Let $(\vec{s}, \xi)$ be the $\mathcal{A}$ configuration reached on w, and let $(\vec{t}, \xi)$ be the $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ configuration reached on w.

Let $\mathtt{q}$ be the role such that $x \in \Sigma_{\mathtt{q}}$, and let s, t denote $\vec{s}_{\mathtt{q}}$, $\vec{t}_{\mathtt{q}}$ from the respective CSM configurations reached on w for $\mathcal{A}$ and $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$.

To show that wx is a trace of $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$, it thus suffices to show that there exists a state t' and a transition $t \xrightarrow{x} t'$ in $\mathscr{C}(\mathbf{G}, \mathtt{q})$.

Since $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ implements $\mathbf{G}$, all finite traces of $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ are prefixes of $\mathcal{L}(\mathbf{G})$. In other words, $w \in \text{pref}(\mathcal{L}(\mathbf{G}))$. Let ρ be a run such that $\rho \in I(w)$; such a run must exist from [38, Lemma 6.3]. Let $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ be the unique splitting of ρ for $\mathtt{q}$ matching w. From the definition of state decoration, it holds that $G \in d(s)$. From the definition of the subset construction, it holds that $G \in t$.

We proceed by case analysis on whether x is a send or receive event.

– Case $x \in \Sigma_{\mathtt{q},!}$. Let $x = \mathtt{q}\triangleright\mathtt{r}!m$. By assumption, there exists $s \xrightarrow{\mathtt{q}\triangleright\mathtt{r}!m} s'$ in $A_{\mathtt{q}}$. We instantiate *Send Decoration Validity* from C_1 with $\mathtt{q}$ and this transition to obtain:

$$\text{tr-orig}(d(s) \xrightarrow{\mathtt{q}\triangleright\mathtt{r}!m} d(s')) = d(s) \ .$$

From $G \in d(s)$, it follows that there exists $G' \in Q_{\mathbf{G}}$ such that $G \xrightarrow{x}{}^{*} G' \in \delta_{\downarrow}$. Because $G \in t$, the existence of t' such that $t \xrightarrow{\mathtt{q}\triangleright\mathtt{r}!m} t'$ is a transition in $\mathscr{C}(\mathbf{G}, \mathtt{p})$ follows immediately from the definition of $\mathscr{C}(\mathbf{G}, \mathtt{q})$'s transition relation.

– Case $x \in \Sigma_{\mathtt{q},?}$. Let $x = \mathtt{q}\triangleleft\mathtt{r}?m$.

From the fact that ρ is a maximal run in $\mathbf{G}$ with unique splitting $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ for $\mathtt{q}$ matching w, it holds that $w\Downarrow_{\Sigma_{\mathtt{q}}} \cdot \mathtt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}} \in \text{pref}(\mathcal{L}(\mathbf{G}))\Downarrow_{\Sigma_{\mathtt{q}}}$. From [38, Lemma 4.3], $\mathcal{L}(\mathbf{G})\Downarrow_{\Sigma_{\mathtt{q}}} = \mathcal{L}(\mathscr{C}(\mathbf{G}, \mathtt{q}))$. Therefore, there exists a t'' such that $t \xrightarrow{\mathtt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}}} t''$ is a transition in $\mathscr{C}(\mathbf{G}, \mathtt{q})$. From *Transition Exhaustivity*, there likewise exists an s'' such that $s \xrightarrow{\mathtt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}}} s''$ is a transition in $A_{\mathtt{q}}$.
We now proceed by showing that it must be the case that $\mathtt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}} = x$. The reasoning closely follows that in [38, Lemma 6.4], which showed that if Receive Validity holds for the subset construction, and some role's subset construction automaton can perform a receive action, then the trace extended with the receive action remains consistent with any global run it was consistent with before. We generalize this property in terms of available message sets in the following lemma, whose proof can be found in the extended version [37].

Lemma 4.13. *Let $\mathcal{A}$ be a CSM, $\mathtt{q}$ be a role, and w, wx be traces of $\mathcal{A}$ such that $x = \mathtt{q}\triangleleft\mathtt{r}?m$. Let s be the state of $\mathtt{q}$'s state machine in the $\mathcal{A}$ configuration reached on w. Let ρ be a run that is consistent with w, i.e. for all $\mathtt{p} \in \mathcal{P}$. $w\Downarrow_{\Sigma_{\mathtt{p}}} \leq \mathtt{split}(\mathtt{trace}(\rho))\Downarrow_{\Sigma_{\mathtt{p}}}$. Let $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ be the unique splitting of ρ for $\mathtt{q}$ matching w. If $\mathtt{r}\triangleright\mathtt{q}!m \notin M^{\mathtt{q}}_{(G' \ldots)}$, then $x = \mathtt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}}$.*

We wish to apply Lemma 4.13 with ρ to conclude that $\texttt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}} = x$. We satisfy the assumption that $\mathtt{r}\triangleright\mathtt{q}!m \notin M^{\mathtt{q}}_{(G' \ldots)}$ by instantiating *Receive Decoration Validity* with $s \xrightarrow{\mathtt{q}\triangleleft\mathtt{r}?m} s'$, $s \xrightarrow{\texttt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}}} s''$ and G'. The fact that $G' \in \text{tr-dest}(d(s) \xrightarrow{\texttt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}}} d(s''))$ follows from the fact that $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ is a run in $\mathbf{G}$ and the definition of state decoration (Definition 4.6). Thus, we conclude from $\texttt{split}(l)\Downarrow_{\Sigma_{\mathtt{q}}} = x$ that there exists a transition $t \xrightarrow{x} t''$ in $\mathscr{C}(\mathbf{G}, \mathtt{q})$.

This concludes our proof that any trace in $\mathcal{A}$ is also a trace of $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$.
Claim 2: $\forall\, w \in \Sigma^*_{async}.\ w$ is terminated in $\mathcal{A} \implies w$ is terminated in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ and w is maximal in $\mathcal{A}$.

Let w be a terminated trace in $\mathcal{A}$. By Claim 1, w is also a trace in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$. Let ξ be the channel configuration uniquely determined by w. Let the $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ configuration reached on w be $(\vec{t}, \xi)$, and let $(\vec{s}, \xi)$ be the $\mathcal{A}$ configuration reached on w. To see that every terminated trace in $\mathcal{A}$ is also terminated in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$, assume by contradiction that w is not terminated in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$. Because $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock-free, there must exist a role that can take a step in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$. Let $\mathtt{q}$ be this role, and let x be the transition that is enabled from $\vec{t}_{\mathtt{q}}$. From *Local Language Inclusion* and the fact that $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock-free, it holds that x is also enabled from $\vec{s}_{\mathtt{q}}$. We arrive at a contradiction. To see that every terminated trace in $\mathcal{A}$ in maximal, from the above we know that w is terminated in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$. From the fact that $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock-free, w is maximal in $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$: all states in $\vec{t}$ are final and all channels in ξ are empty. From *Local Language Inclusion*, it follows that all states in $\vec{s}$ are also final, and thus w is maximal in $\mathcal{A}$. □

Lemma 4.14 (Completeness of C_1). *If $\mathcal{A}$ and $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ are equivalent, then C_1 holds.*

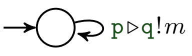
We show completeness via modus tollens: we assume a violation in C_1 and the fact that $\mathcal{A}$ and $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ are equivalent, and prove a contradiction. Since C_1 is a conjunction of four conditions, we derive a contradiction from the violation of each condition in turn. In the interest of proof reuse, we specify which of the two refinement conjuncts we contradict for each condition, and refer the reader to the extended version [37] for the full proofs.

From the negation of *Transition Exhaustivity* and *Final State Validity*, we contradict the fact that $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ refines $\mathcal{A}$.

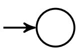
Lemma 4.15. *If $\mathcal{A}$ violates* Transition Exhaustivity *or* Final State Validity*, then it does not hold that $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ refines $\mathcal{A}$.*

Unlike the proofs for *Transition Exhaustivity* and *Final State Validity*, the proofs for the remaining two conditions require both refinement conjuncts to prove a contradiction. Both proofs find a contradiction by obtaining a witness from the violation of *Send Decoration Validity* and *Receive Decoration Validity* respectively, and showing that the same witness can be used to refute Send and Receive Validity for the subset construction.

Lemma 4.16. *If $\mathcal{A}$ violates* Send Decoration Validity *or* Receive Decoration Validity*, then it does not hold that $\mathcal{A}$ and $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ are equivalent.*



(a) State machine $\mathscr{C}(\mathbf{G}, \mathtt{p})$



(b) State machine $B'_{\mathtt{q}}$

Fig. 4: CSM violating subprotocol fidelity with respect to $\mathbf{G}_{loop}$

5 Deciding Protocol Refinement

We now turn our attention to *Protocol Refinement*, which asks when an implementation can safely substitute another in all well-behaved contexts with respect to $\mathbf{G}$. Here, we introduce a new notion of refinement with respect to a global type.

Definition 5.1 (Protocol refinement with respect to G). *We say that a CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ refines a CSM $\{\!\{B_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ with respect to a global type $\mathbf{G}$ if the following properties hold: (i)* subprotocol fidelity: $\exists S \subseteq \mathcal{L}(\mathsf{GAut}(\mathbf{G})).\ \mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{C}^{\sim}(\mathtt{split}(S))$, *(ii)* language inclusion: $\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) \subseteq \mathcal{L}(\{\!\{B_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$*, and (iii)* deadlock freedom: $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *is deadlock-free.*

Item i, *subprotocol fidelity*, sets our notion of refinement apart from standard refinement. We motivate this difference briefly using an example. Consider the CSM consisting of the subset construction for $\mathtt{p}$ and $B'_{\mathtt{q}}$, depicted in Fig. 4. This CSM recognizes only words of the form $(\mathtt{p}\triangleright\mathtt{q}!m)^{\omega}$. It is nonetheless considered to refine the global type $\mathbf{G}_{loop} := \mu t.\ \mathtt{p} \to \mathtt{q} : m.\ t$ according to the standard notion of refinement, despite the fact that $\mathtt{p}$'s messages are never received by $\mathtt{q}$. This is because $\mathcal{L}(\mathbf{G}_{loop})$, containing only infinite words, is defined in terms of an asymmetric downward closure operator $\preceq^{\omega}_{\sim}$, which allows receives to be infinitely postponed. We desire a notion of refinement that allows roles to select which runs to follow in a global type, but disallows them from selecting which words to implement among ones that follow the same run. More formally, our notion of protocol refinement prohibits selectively implementing words that are equivalent under the indistinguishability relation $\sim$: any CSM that refines another with respect to a global type has a language that is closed under $\sim$.

In the remainder of the paper, we refer to refinement with respect to $\mathbf{G}$, and omit mention of $\mathbf{G}$ when clear from context. Again using the fact that $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is an implementation for $\mathbf{G}$, we say that a CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ refines $\mathbf{G}$ if it refines $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$.

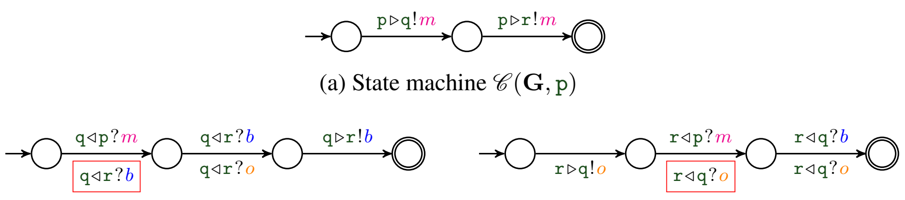
We motivate our formulation of the *Protocol Refinement* problem by posing the following variation of *Protocol Verification*, which we call *Monolithic Protocol Refinement*:

Given an implementable global type $\mathbf{G}$ and a CSM $\mathcal{A}$, does $\mathcal{A}$ *refine* $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$?

This variation asks for a condition, C'_1, that satisfies the equivalence:

$$C'_1 \Leftrightarrow \mathcal{A} \text{ refines } \{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}.$$

Clearly, C_1 is still a sound candidate as equivalence of two CSMs implies bidirectional protocol refinement. It is instructive to analyze why the completeness arguments for C_1 fail. Recall that the completeness proofs for *Send Decoration Validity*



(a) State machine $\mathscr{C}(\mathbf{G}, \mathtt{p})$

(b) State machine $A'_{\mathtt{q}}$

(c) State machine $A'_{\mathtt{r}}$

Fig. 5: Subset construction for $\mathtt{p}$ and two state machines for $\mathtt{q}$ and $\mathtt{r}$ for $\mathbf{G}'$

and *Receive Decoration Validity* used the violation of each condition to obtain a local state with a non-empty decoration set, which in turn gives rise to a prefix in $\mathcal{L}(\mathbf{G})$ that must be a trace in the subset construction. This trace is then replayed in the arbitrary CSM, extended in the arbitrary CSM, and then replayed again in the subset construction. This sequence of replaying arguments critically relied on both the assumption that $\mathcal{A}$ refines $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$, and the assumption that $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ refines $\mathcal{A}$.

If we cannot assume that $\mathcal{A}$ recognizes every behavior of $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$, then the reachable local states of $\mathcal{A}$ are no longer precisely characterized by having a non-empty decoration set.

Consider the example global type $\mathbf{G}'$:

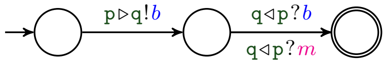
$$\mathbf{G}' := \mathtt{p} \to \mathtt{q} : m. + \begin{cases} \mathtt{r} \to \mathtt{q} : b. \, \mathtt{p} \to \mathtt{r} : m. + \begin{cases} \mathtt{q} \to \mathtt{r} : b. \, 0 \\ \mathtt{q} \to \mathtt{r} : o. \, 0 \end{cases} \\ \mathtt{r} \to \mathtt{q} : o. \, \mathtt{p} \to \mathtt{r} : m. + \begin{cases} \mathtt{q} \to \mathtt{r} : b. \, 0 \\ \mathtt{q} \to \mathtt{r} : o. \, 0 \end{cases} \end{cases}$$

Let the CSM $\mathcal{A}'$ consist of the subset construction automaton for $\mathtt{p}$, and the state machines $A'_{\mathtt{q}}$ and $A'_{\mathtt{r}}$, given in Figs. 5b and 5c. The receive transitions highlighted in red are safe despite violating *Receive Decoration Validity*, because $\mathtt{q}$ and $\mathtt{r}$ coordinate with each other on which runs of $\mathbf{G}$ they eliminate: $\mathtt{r}$ chooses to never send a b to $\mathtt{q}$, thus $\mathtt{q}$'s highlighted transition is safe, and conversely, $\mathtt{q}$ never chooses to send o to $\mathtt{r}$, thus $\mathtt{r}$'s highlighted transition is safe. Consequently, $\mathcal{A}'$ refines $\mathbf{G}'$ despite violating C_1.

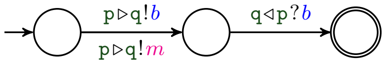
This example shows that any condition C'_1 that is compositional must sacrifice completeness. In fact, deciding whether an arbitrary CSM $\mathcal{A}$ refines the subset construction $\{\!\{\mathscr{C}(\mathbf{G}, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ for some global type $\mathbf{G}$ can be shown to be PSPACE-hard via a reduction from the deadlock-freedom problem for 1-safe Petri nets [24]. We refer the reader to the extended version [37] for the full construction.

Lemma 5.2. *The* Monolithic Protocol Refinement *problem is PSPACE-hard.*

Fortunately, we can recover completeness and tractability by only allowing changes to one state machine in $\mathcal{A}$ at a time. Next, we formalize the notions of *CSM contexts* and *well-behavedness* with respect to $\mathbf{G}$. We use $\mathcal{A}[\cdot]_{\mathtt{p}}$ to denote a CSM context with a hole for role $\mathtt{p} \in \mathcal{P}$, and $\mathcal{A}[A]_{\mathtt{p}}$ to denote the CSM obtained by instantiating the context with state machine A for $\mathtt{p}$. We define well-behaved contexts in terms of the canonical implementation $\mathscr{C}(\mathbf{G}, \mathtt{p})$.



(a) Removing sends



(b) Removing receives

Fig. 6: Two candidate implementations for $\mathtt{p}$

Definition 5.3 (Well-behaved CSM contexts with respect to G). *Let $\mathcal{A}[\cdot]_{\mathtt{p}}$ be a CSM context. We say that $\mathcal{A}[\cdot]_{\mathtt{p}}$ is well-behaved with respect to $\mathbf{G}$ if $\mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}$ refines $\mathbf{G}$. We omit $\mathbf{G}$ when clear from context.*

Protocol Refinement asks to find a C_2 that satisfies the following:

Theorem 5.4. *Let $\mathbf{G}$ be an implementable global type, $\mathtt{p}$ be a role, and A, B be state machines for role $\mathtt{p}$ such that for all well-behaved contexts $\mathcal{A}[\cdot]_{\mathtt{p}}$, $\mathcal{A}[B]_{\mathtt{p}}$ refines $\mathbf{G}$. Then, for all well-behaved contexts $\mathcal{A}[\cdot]_{\mathtt{p}}$, $\mathcal{A}[A]_{\mathtt{p}}$ refines $\mathcal{A}[B]_{\mathtt{p}}$ if and only if C_2 is satisfied.*

5.1 Protocol Refinement Relative to Subset Construction

As a stepping stone, we first consider the special case of *Protocol Refinement* when B is the subset construction automaton for role $\mathtt{p}$. That is, we present C_2' that satisfies the following equivalence:

$$C_2' \Leftrightarrow \text{for all well-behaved contexts } \mathcal{A}[\cdot]_{\mathtt{p}},\ \mathcal{A}[A]_{\mathtt{p}} \text{ refines } \mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}.$$

The relaxation on language equality from *Protocol Verification* means that state machine A no longer needs to satisfy *Local Language Inclusion*, which grants us more flexibility: state machines are now permitted to remove send events. Let us revisit our example global type, $\mathbf{G}_1$:

$$\mathbf{G}_1 := + \begin{cases} \mathtt{p}\to\mathtt{q}:b.\ \mathtt{q}\to\mathtt{p}:b.\ 0 \\ \mathtt{p}\to\mathtt{q}:m.\ \mathtt{q}\to\mathtt{p}:m.\ 0 \end{cases}$$

Consider the candidate state machine for role $\mathtt{p}$ given in Fig. 6a. The CSM obtained from inserting this state machine into any well-behaved context refines $\mathbf{G}$, despite the fact that $\mathtt{p}$ never sends m. In general, send events can safely be removed from reachable states in a local state machine without violating subprotocol fidelity or deadlock freedom, as long as *not all* of them are removed.

The same is not true of receive events, on the other hand. The state machine in Fig. 6b is not a safe candidate for $\mathtt{p}$, because it causes a deadlock in the well-behaved context that consists of the subset construction for every other role.

Our characterization intuitively follows the notion that input types (receive events) are covariant, and output types (send events) are contravariant. However, note that the state machine above cannot be represented in existing works [8, 20, 26]: their local types support neither states with both outgoing send and receive events, nor states with outgoing send or receive events to/from different roles.

Our characterization C_2' reuses *Send Decoration Validity*, *Receive Decoration Validity* and *Final State Validity* from C_1, but splits *Transition Exhaustivity* into a separate condition for send and receive events, to reflect the aforementioned asymmetry between them.

Definition 5.5 **(C_2').** *Let* $\mathtt{p} \in \mathcal{P}$ *be a role and let* $A = (Q, \Sigma_{\mathtt{p}}, s_0, \delta, F)$ *be a state machine for* $\mathtt{p}$. C_2' *is satisfied when the following conditions hold in addition to* Send Decoration Validity*,* Receive Decoration Validity *and* Final State Validity*:*

- Send Preservation*: every state containing a send-originating global state must have at least one outgoing send transition:*
  $\forall s \in Q.\ \exists G \in Q_{\mathbf{G},!}.\ G \in d(t) \implies \exists x \in \Sigma_{\mathtt{p},!},\ s' \in Q.\ s \xrightarrow{x} s' \in \delta.$
- Receive Exhaustivity*: every receive transition that is enabled in some global state decorating* s *must be an outgoing transition from* s*:*
  $\forall s \in Q.\ \forall G \xrightarrow{x}{}^{*} G' \in \delta_{\downarrow}.\ G \in d(s) \wedge x \in \Sigma_{\mathtt{p},?} \implies \exists s' \in Q.\ s \xrightarrow{x} s' \in \delta.$

We want to show the following equivalence:

$$C_2' \Leftrightarrow \text{for all well-behaved contexts } \mathcal{A}[\cdot]_{\mathtt{p}},\ \mathcal{A}[A]_{\mathtt{p}} \text{ refines } \mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}.$$

We first prove the soundness of C_2'.

Lemma 5.6 **(Soundness of C_2').** *If* C_2' *holds, then for all well-behaved contexts* $\mathcal{A}[\cdot]_{\mathtt{p}}$*,* $\mathcal{A}[A]_{\mathtt{p}}$ *refines* $\mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}$.

Proof. Let $\mathcal{A}[\cdot]_{\mathtt{p}}$ be a well-behaved context with respect to $\mathbf{G}$. Like before, we first prove that any trace in $\mathcal{A}[A]_{\mathtt{p}}$ is a trace in $\mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}$.
Claim 1: $\forall\, w \in \Sigma_{async}^{\infty}.\ w$ is a trace in $\mathcal{A}[A]_{\mathtt{p}} \implies w$ is a trace in $\mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}$.

The proof of Claim 1 for C_2' differs from that for C_1 in only two ways. We discuss the differences in detail below, and avoid repeating the rest of the proof.

1. C_1 grants that every role's state machine satisfies *Send Decoration Validity* and *Receive Decoration Validity*, whereas C_2 only guarantees the conditions for role $\mathtt{p}$. Correspondingly, $\mathcal{A}[A]_{\mathtt{p}}$ only differs from $\mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}$ in $\mathtt{p}$'s state machine; all other roles' state machines are identical between the two CSMs. Therefore, the induction step requires a case analysis on the role whose alphabet the event x belongs to. In the case that $x \in \Sigma_{\mathtt{q}}$ where $\mathtt{q} \neq \mathtt{p}$, the induction hypothesis is trivially re-established by the fact that $\mathtt{q}$'s state machine is identical in both CSMs. In the case that $x \in \Sigma_{\mathtt{p}}$, we proceed to reason that x can also be performed by $\mathscr{C}(\mathbf{G},\mathtt{p})$ in the same well-behaved context.
2. C_1 includes *Transition Exhaustivity*, which allows us to conclude that given a run with unique splitting $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ for $\mathtt{p}$ matching w and the fact that $G \in s$, there must exist a transition $s \xrightarrow{\mathtt{split}(l)\Downarrow_{\Sigma_{\mathtt{p}}}} s''$ in $\mathtt{p}$'s state machine. Lemma 4.13 can then be instantiated directly with $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ to complete the proof. C_2, on the other hand, splits *Transition Exhaustivity* into *Send Preservation* and *Receive Exhaustivity*, and we can only establish that such a transition exists and reuse the proof in the case that $\mathtt{split}(l)\Downarrow_{\Sigma_{\mathtt{p}}} \in \Sigma_{\mathtt{p},?}$. Since A is permitted to remove send

events, if $\texttt{split}(l){\Downarrow}_{\Sigma_\texttt{p}} \in \Sigma_{\texttt{p},!}$, the transition $s \xrightarrow{\texttt{split}(l){\Downarrow}_{\Sigma_\texttt{p}}} s''$ may not exist at all in A. However, the existence of a run $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ where l is a send event for $\texttt{p}$ makes G a send-originating global state in $\texttt{p}$'s projection by erasure automaton. *Send Preservation* thus guarantees that there exists a transition $s \xrightarrow{x'} s'''$ in A such that $x' \in \Sigma_{\texttt{p},!}$. By *Send Decoration Validity*, x' originates from G in the projection by erasure, and we can find another run ρ' such that $\alpha' \cdot G \xrightarrow{l'} G'' \cdot \beta'$ is the unique splitting for $\texttt{p}$ matching w and $\texttt{split}(l'){\Downarrow}_{\Sigma_\texttt{p}} = x'$. We satisfy the assumption that $\texttt{r}\triangleright\texttt{p}!m \notin M^{\texttt{p}}_{(G''\ldots)}$ by instantiating *Receive Decoration Validity* with $\texttt{p}$, $s \xrightarrow{x} s'$, $s \xrightarrow{\texttt{split}(l'){\Downarrow}_{\Sigma_\texttt{p}}} s''$ and G''. The fact that $G'' \in \text{tr-dest}(d_{\mathbf{G}}(s) \xrightarrow{\texttt{split}(l'){\Downarrow}_{\Sigma_\texttt{p}}} d_{\mathbf{G}}(s''))$ follows from the fact that $\alpha \cdot G \xrightarrow{l'} G'' \cdot \beta'$ is a run in $\mathbf{G}$ and Definition 4.6. Instantiating Lemma 4.13 with ρ', we obtain $\texttt{split}(l'){\Downarrow}_{\Sigma_\texttt{p}} = x$, which is a contradiction: x is a receive event and $\texttt{split}(l'){\Downarrow}_{\Sigma_\texttt{p}}$ is a send event. Thus, it cannot be the case that $\texttt{split}(l'){\Downarrow}_{\Sigma_\texttt{p}} \in \Sigma_{\texttt{p},!}$.

This concludes our proof that any trace in $\mathcal{A}[A]_\texttt{p}$ is also a trace in $\mathcal{A}[\mathscr{C}(\mathbf{G},\texttt{p})]_\texttt{p}$.

The following claim completes our soundness proof:

Claim 2: $\forall\, w \in \Sigma^*_{async}.\ w$ is terminated in $\mathcal{A}[A]_\texttt{p} \implies w$ is terminated in $\mathcal{A}[\mathscr{C}(\mathbf{G},\texttt{p})]_\texttt{p}$ and w is maximal in $\mathcal{A}[A]_\texttt{p}$.

The proof of Claim 2 for C_1 again relies on *Local Language Inclusion*, which is unavailable to C_2'. Instead, we turn to *Send Preservation*, *Receive Exhaustivity* and *Final State Validity* to establish this claim. Let w be a terminated trace in $\mathcal{A}[A]_\texttt{p}$. By Claim 1, it holds that w is a trace in $\mathcal{A}[\mathscr{C}(\mathbf{G},\texttt{p})]_\texttt{p}$. Let ξ be the channel configuration uniquely determined by w. Let $(\vec{s},\xi)$ be the $\mathcal{A}[\mathscr{C}(\mathbf{G},\texttt{p})]_\texttt{p}$ configuration reached on w, and let $(\vec{t},\xi)$ be the $\mathcal{A}[A]_\texttt{p}$ configuration reached on w. To see that w is terminated in $\mathcal{A}[\mathscr{C}(\mathbf{G},\texttt{p})]_\texttt{p}$, suppose by contradiction that w is not terminated in $\mathcal{A}[\mathscr{C}(\mathbf{G},\texttt{p})]_\texttt{p}$. Because $\mathcal{A}[\mathscr{C}(\mathbf{G},\texttt{p})]_\texttt{p}$ is deadlock-free, and because the state machines for all non-$\texttt{p}$ roles are identical between the two CSMs, it must be the case that $\texttt{p}$ witnesses the non-termination of w, in other words, $\mathscr{C}(\mathbf{G},\texttt{p})$ can take a transition that A cannot. Let $\vec{s}_\texttt{p} \xrightarrow{x} s'$ be the transition that $\texttt{p}$ can take from $\vec{s}_\texttt{p}$. Let G be a state in $\vec{s}_\texttt{p}$; such a state is guaranteed to exist by the fact that no reachable states in the subset construction are empty. Then, in the projection by erasure automaton, the initial state reaches G on $w{\Downarrow}_{\Sigma_\texttt{p}}$. By the fact that w is a trace of $\mathcal{A}[A]_\texttt{p}$, it holds that s_0 reaches $\vec{s}_\texttt{p}$ on $w{\Downarrow}_{\Sigma_\texttt{p}}$ in A. By the definition of state decoration, $G \in d(\vec{t}_\texttt{p})$.

- If $x \in \Sigma_!$, it follows that G is a send-originating global state. By *Send Preservation*, for any state in A that contains at least one send-originating global state, of which $\vec{t}_\texttt{p}$ is one, there exists a transition $\vec{t}_\texttt{p} \xrightarrow{x'} t'$ such that $x' \in \Sigma_{\texttt{p},!}$. Because send transitions in a CSM are always enabled, role $\texttt{p}$ can take this transition in $\mathcal{A}[A]_\texttt{p}$. We reach a contradiction to the fact that w is terminated in $\mathcal{A}[A]_\texttt{p}$.
- If $x \in \Sigma_?$, it follows that G is a receive-originating global state. From *Receive Exhaustivity*, any receive event that originates from any global state in $d(\vec{t}_\texttt{p})$ must also originate from $\vec{t}_\texttt{p}$. Therefore, there must exist t' such that $\vec{t}_\texttt{p} \xrightarrow{x} t'$ is a transition in $B'_\texttt{p}$. Because the channel configuration is identical in both CSMs, role $\texttt{p}$ can

take this transition in $\mathcal{A}[A]_{\mathtt{p}}$. We again reach a contradiction to the fact that w is terminated in $\mathcal{A}[A]_{\mathtt{p}}$.

To see that w is maximal in $\mathcal{A}[A]_{\mathtt{p}}$, observe that for all roles $\mathtt{q} \neq \mathtt{p}$, $\vec{s}_{\mathtt{q}} = \vec{t}_{\mathtt{q}}$. Thus, it remains to show that $\vec{t}_{\mathtt{p}}$ is a final state in A. Because $\vec{s}_{\mathtt{p}}$ is a final state, by the definition of the subset construction there exists a global state $G \in \vec{s}_{\mathtt{p}}$ such that the projection erasure automaton reaches G on $w{\Downarrow}_{\Sigma_{\mathtt{p}}}$ and G is a final state. Because A reaches $\vec{t}_{\mathtt{p}}$ on $w{\Downarrow}_{\Sigma_{\mathtt{p}}}$, by Definition 4.6 it holds that $G \in d(\vec{t}_{\mathtt{p}})$. By *Final State Validity*, it holds that $\vec{t}_{\mathtt{p}}$ is a final state in A. This concludes our proof that any terminated trace in $\mathcal{A}[A]_{\mathtt{p}}$ is also a terminated trace in $\mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}$, and is maximal in $\mathcal{A}[A]_{\mathtt{p}}$.

Together, Claim 1 and 2 establish that $\mathcal{A}[A]_{\mathtt{p}}$ satisfies language inclusion (Item ii) and deadlock freedom (Item iii). It remains to show that $\mathcal{A}[A]_{\mathtt{p}}$ satisfies subprotocol fidelity (Item i). This follows immediately from [40, Lemma 22], which states that all CSM languages are closed under the indistinguishability relation $\sim$. □

Lemma 5.7 (Completeness of C_2'). *If for all well-behaved contexts $\mathcal{A}[\cdot]_{\mathtt{p}}$, $\mathcal{A}[A]_{\mathtt{p}}$ refines $\mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}$, then C_2' holds.*

As before, we prove the modus tollens of this implication, which states that if C_2' does not hold, then there exists a well-behaved context $\mathcal{A}[\cdot]_{\mathtt{p}}$ such that $\mathcal{A}[A]_{\mathtt{p}}$ does not protocol-refine $\mathcal{A}[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}}$.

We first turn our attention to finding a well-behaved witness context $\mathcal{A}[\cdot]_{\mathtt{p}}$ such that we can refute subprotocol fidelity, language inclusion, or deadlock freedom. It turns out that the context consisting of the subset construction automaton for every other role is a suitable witness. We denote this context by $\mathscr{C}(\mathbf{G})[\cdot]_{\mathtt{p}}$ and note that it is trivially well-behaved because $\mathscr{C}(\mathbf{G})[\mathscr{C}(\mathbf{G},\mathtt{p})]_{\mathtt{p}} = \{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$.

Recall from the completeness arguments for C_1 that we obtained a violating state in some state machine A with a non-empty decoration set from the negation of each condition in C_1. From this state's decoration set we obtained a witness global state G, and in turn a run $\alpha \cdot G$ in $\mathbf{G}$, and from the assumption that $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ refines $\mathcal{A}$, we argued that $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$ is a trace in $\mathcal{A}$. We then showed that A is in the violating state in the $\mathcal{A}$ configuration reached on $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$, and from there we used each violated condition to find a contradiction.

The completeness proof for C_2' cannot similarly use the fact that $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ refines $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$. Instead, we must separately establish that every state with a non-empty decoration set can be reached on a trace shared by both $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$ and $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$. The following lemma achieves this:

Lemma 5.8. *Let A be a state machine for $\mathtt{p}$ and s be a state in A. Let $G \in d(s)$, and let $u \in \Sigma_{\mathtt{p}}^*$ be a word such that $s_0 \xrightarrow{u}{}^* s$ in A. Then, there exists a run $\alpha \cdot G$ of $\mathsf{GAut}(\mathbf{G})$ such that $\mathtt{split}(\mathtt{trace}(\alpha \cdot G)){\Downarrow}_{\Sigma_{\mathtt{p}}} = u$, $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$ is a trace in $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$ and in the CSM configuration reached on $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$, A is in state s.*

With Lemma 5.8 replacing the assumption that $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ refines $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$, we can reuse the construction in Lemma 4.16 to obtain a word that is a trace in $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$ but not a trace in $\{\!\{\mathscr{C}(\mathbf{G},\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$, thus evidencing the necessity of *Send Decoration Validity* and *Receive Decoration Validity*. The proof of Lemma 5.9 proceeds identically to that of Lemma 4.16 and is thus omitted.

Lemma 5.9. *If A violates* Send Decoration Validity *or* Receive Decoration Validity, *then it does not hold that for all well-behaved contexts $\mathcal{A}[\cdot]_{\mathtt{p}}$, $\mathcal{A}[A]_{\mathtt{p}}$ refines $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$.*

We also use Lemma 5.8 to show the necessity of *Send Preservation*, *Receive Exhaustivity* and *Final State Validity*. As a starting point, let A, s, u and $\alpha \cdot G$ be obtained from Lemma 5.8 and the violation of *Send Preservation*. To show the necessity of *Send Preservation*, we consider the largest extension v of u in $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$. In the case that u is terminated in $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$, we refute deadlock freedom from the fact that u is not maximal: $G \in s$ is a send-originating state, and final states in $\mathsf{GAut}(\mathbf{G})$ do not contain outgoing transitions. If $v \neq u$, there exists a run $\alpha \cdot G \xrightarrow{\mathtt{p}\rightarrow\mathtt{q}:m} G' \cdot \beta$ such that $\mathtt{split}(\mathtt{trace}(\alpha \cdot G \xrightarrow{\mathtt{p}\rightarrow\mathtt{q}:m} G' \cdot \beta) {\Downarrow_{\Sigma_{\mathtt{p}}}} = v{\Downarrow_{\Sigma_{\mathtt{p}}}}$. By subprotocol fidelity, $\mathtt{split}(\mathtt{trace}(\alpha \cdot G \xrightarrow{\mathtt{p}\rightarrow\mathtt{q}:m} G' \cdot \beta))$ is a trace in $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$. Consequently, $\mathtt{split}(\mathtt{trace}(\alpha \cdot G \xrightarrow{\mathtt{p}\rightarrow\mathtt{q}:m} G' \cdot \beta)){\Downarrow_{\Sigma_{\mathtt{p}}}}$ is a prefix in A. We find a contradiction from the fact that A is deterministic and there is no outgoing transition labeled $\mathtt{p}\triangleright\mathtt{q}!m$ from s. Similar arguments can be used to show the necessity of *Receive Exhaustivity*. Finally, for *Final State Validity*, in the case that s is non-final in A but contains a final state in $\mathsf{GAut}(\mathbf{G})$, we can instantiate Lemma 5.8 with this final state and show that u evidences a deadlock.

Lemma 5.10. *If A violates* Send Preservation, Receive Exhaustivity *or* Final State Validity, *then it does not hold that for all well-behaved contexts $\mathcal{A}[\cdot]_{\mathtt{p}}$, $\mathcal{A}[A]_{\mathtt{p}}$ refines $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$.*

5.2 Protocol Refinement (General Case)

Equipped with the solution to a special case, we are ready to revisit the general case of *Protocol Refinement*, which asks to find a C_2 that satisfies the following:

$$C_2 \Leftrightarrow \text{for all well-behaved contexts } \mathcal{A}[\cdot]_{\mathtt{p}},\ \mathcal{A}[A]_{\mathtt{p}} \text{ refines } \mathcal{A}[B]_{\mathtt{p}}.$$

Critical to the former problems is the fact that the state decoration function precisely captures those states in a local state machine that are reachable in some CSM execution, under some assumptions on the context: a state is reachable if and only if its decoration set is non-empty. This allows the conditions in C_1 and C_2' to precisely characterize the reachable local states.

The second problem generalizes the subset projection to an arbitrary state machine B, and asks whether a candidate state machine A (the subtype) refines B (the supertype) in any well-behaved context. Unfortunately, we cannot simply decorate the subtype with the supertype's states, because not all states in the supertype are reachable. Instead, we need to restrict the set of states in the supertype to those that themselves have non-empty decoration sets with respect to $\mathbf{G}$.

In the remainder of this section, let $\mathtt{p} \in \mathcal{P}$ be a role, let $B = (Q_B, \Sigma_{\mathtt{p}}, t_0, \delta_B, F_B)$ denote the supertype state machine for $\mathtt{p}$, and let $A = (Q_A, \Sigma_{\mathtt{p}}, s_0, \delta_A, F_A)$ denote the subtype state machine for $\mathtt{p}$. We modify our state decoration function in Definition 4.6 to map states of A to subsets of states in B that themselves have non-empty decoration sets with respect to $\mathbf{G}$.

Definition 5.11 (State decoration with respect to a supertype). *Let* $\mathbf{G}$ *be a global type. Let* $\mathtt{p} \in \mathcal{P}$ *be a role, and let further* $B = (Q_B, \Sigma_{\mathtt{p}}, t_0, \delta_B, F_B)$ *and* $A = (Q_A, \Sigma_{\mathtt{p}}, s_0, \delta_A, F_A)$ *be two deterministic finite state machines for* $\mathtt{p}$*. We define a total function* $d_{\mathbf{G},B,A} : Q' \to 2^Q$ *that maps each state in* A *to a subset of states in* B *such that:*

$$d_{\mathbf{G},B,A}(s) = \{t \in Q_B \mid \exists u \in \Sigma_{\mathtt{p}}{}^*.\ s_0 \xrightarrow{u}{}^* s \in \delta_A \wedge t_0 \xrightarrow{u}{}^* t \in \delta_B \wedge d(t) \neq \emptyset\}$$

We again omit the subscripts $\mathbf{G}$ and A when clear from context, but retain the subscript B to distinguish d_B from d in Definition 4.6.

We likewise require a generalization of tr-orig and tr-dest to be defined in terms of B, instead of the projection by erasure automaton for $\mathtt{p}$.

Definition 5.12 (Transition origin and destination with respect to a supertype). *Let* $\mathbf{G}$ *be a global type, and let* $B = (Q_B, \Sigma_{\mathtt{p}}, t_0, \delta_B, F_B)$ *be a state machine. For* $x \in \Sigma_{\mathtt{p}}$ *and* $s, s' \subseteq Q_B$*, we define the set of* transition origins $\text{tr-orig}(s \xrightarrow{x} s')$ *and* transition destinations $\text{tr-dest}(s \xrightarrow{x} s')$ *as follows:*

$$\text{tr-orig}_B(s \xrightarrow{x} s') := \{t \in s \mid \exists t' \in s'.\, t \xrightarrow{x}{}^* t' \in \delta_B\} \ \textit{and}$$
$$\text{tr-dest}_B(s \xrightarrow{x} s') := \{t' \in s' \mid \exists t \in s.\, t \xrightarrow{x}{}^* t' \in \delta_B\} \ .$$

We present C_2 in terms of the newly defined decoration function d_B.

Definition 5.13 (C_2). *Let* $\mathbf{G}$ *be a global type,* $\mathtt{p} \in \mathcal{P}$ *be a role, and let further* $B = (Q_B, \Sigma_{\mathtt{p}}, t_0, \delta_B, F_B)$ *and* $A = (Q_A, \Sigma_{\mathtt{p}}, s_0, \delta_A, F_A)$ *be two deterministic state machines for* $\mathtt{p}$*.* C_2 *is the conjunction of the following conditions:*

- Send Decoration Subtype Validity*: every send transition* $s \xrightarrow{x} s' \in \delta_A$ *must be enabled in all states of* B *decorating* s*:*
  $\forall s \xrightarrow{\mathtt{p} \triangleright \mathtt{q}!m} s' \in \delta_A.\ \text{tr-orig}_B(d_B(s) \xrightarrow{\mathtt{p} \triangleright \mathtt{q}!m} d_B(s')) = d_B(s).$

- Receive Decoration Subtype Validity*: no receive transition is enabled in an alternative continuation originating from the same state:*
  $\forall s \xrightarrow{\mathtt{p} \triangleleft \mathtt{q}_1 ?m_1} s_1,\ s \xrightarrow{x} s_2 \in \delta_A.\ x \neq \mathtt{p} \triangleleft \mathtt{q}_1 ?\_ \implies$
  $\forall G \in \bigcup_{t \in d_B(s_2)} \{d(t) \mid t \in \text{tr-dest}_B(d_B(s) \xrightarrow{x} d_B(s_2))\}.\ \mathtt{q}_1 \triangleright \mathtt{p}!m_1 \notin M^{\mathtt{p}}_{(G\ldots)}.$

- Send Subtype Preservation*: every state decorated by a send-originating global state must have at least one outgoing send transition:*
  $\forall s \in Q_A.\, (\bigcup_{t \in d_B(s)} d(t) \cap Q_{\mathbf{G},!} \neq \emptyset) \implies \exists x \in \Sigma_{\mathtt{p},!},\ s' \in Q_A.\ s \xrightarrow{x} s' \in \delta_A.$

- Receive Subtype Exhaustivity*: every receive transition that is enabled in some global state decorating s must be an outgoing transition from s:*
  $\forall s \in Q_A.\, \forall G \xrightarrow{x}{}^* G' \in \delta_\downarrow.\ G \in \bigcup_{t \in d_B(s)} d(t) \implies \exists s' \in Q_A.\ s \xrightarrow{x} s' \in \delta_A.$

- Final State Validity*: a reachable state is final if its decorating set contains a final global state:*
  $\forall s \in Q_A.\ \bigcup_{t \in d_B(s)} d(t) \neq \emptyset \implies (\bigcup_{t \in d_B(s)} d(t) \cap F_{\mathbf{G}} \neq \emptyset) \implies s \in F_A.$

We want to show the following equivalence to prove Theorem 5.4:

$$C_2 \Leftrightarrow \text{for all well-behaved contexts } \mathcal{A}[\cdot]_{\mathtt{p}},\ \mathcal{A}[A]_{\mathtt{p}} \text{ refines } \mathcal{A}[B]_{\mathtt{p}}.$$

Lemma 5.14 (Soundness of C_2). *If C_2 holds, then for all well-behaved contexts $\mathcal{A}[\cdot]_{\mathtt{p}}$, $\mathcal{A}[A]_{\mathtt{p}}$ refines $\mathcal{A}[B]_{\mathtt{p}}$.*

Predictably, the proof of soundness is directly adapted from the proof for C_2' by applying suitable "liftings", and can be found in the extended version [37].

Lemma 5.15 (Completeness of C_2). *If for all well-behaved contexts $\mathcal{A}[\cdot]_{\mathtt{p}}$, $\mathcal{A}[A]_{\mathtt{p}}$ refines $\mathcal{A}[B]_{\mathtt{p}}$, then C_2 holds.*

Again, we prove the modus tollens of this implication, and we again are required to find a witness well-behaved context $\mathcal{A}[\cdot]_{\mathtt{p}}$, such that $\mathcal{A}[A]_{\mathtt{p}}$ does not refine $\mathcal{A}[B]_{\mathtt{p}}$ under the assumption of the negation of C_2. In the special case where B is the subset construction automaton, we observed that any state in A with a non-empty decoration set with respect to $\mathbf{G}$ is reachable by the CSM consisting of A and the subset construction context, denoted $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$. We were therefore able to use $\mathscr{C}(\mathbf{G})[\cdot]_{\mathtt{p}}$ as the witness well-behaved context. A similar characterization is true in the general case: a state in A is reachable by $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$ if it has a non-empty decoration set with respect to B. This in turn depends on the fact that we only label states in A with states in B that themselves have non-empty decorating sets with respect to $\mathbf{G}$. The following lemma lifts Lemma 5.8 to the general problem setting:

Lemma 5.16. *Let A, B be two state machines for $\mathtt{p}$, such that for all well-behaved contexts $\mathcal{A}[\cdot]_{\mathtt{p}}$, $\mathcal{A}[B]_{\mathtt{p}}$ refines $\mathbf{G}$. Let s be a state in A, and let t be a state in B such that $t \in d_B(s)$. Let $u \in \Sigma_{\mathtt{p}}^*$ be a word such that $s_0 \xrightarrow{u}{}^* s$ in A. Then, there exists a run $\alpha \cdot G$ of $\mathsf{GAut}(\mathbf{G})$ such that $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))\Downarrow_{\Sigma_{\mathtt{p}}} = u$, $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$ is a trace in both $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$ and $\mathscr{C}(\mathbf{G})[B]_{\mathtt{p}}$ and in the CSM configuration reached on $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$, A is in state s.*

Proof. From the fact that $t \in d_B(s)$ and the definition of state decoration (Definition 5.11), it holds that $d(t) \neq \emptyset$ and $t_0 \xrightarrow{u}{}^* t \in \delta_B$. Let $G \in d(t)$. We apply Lemma 5.8 to obtain a run $\alpha \cdot G$ such that $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))\Downarrow_{\Sigma_{\mathtt{p}}} = u$, $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$ is a trace in $\mathscr{C}(\mathbf{G})[B]_{\mathtt{p}}$ and in the $\mathscr{C}(\mathbf{G})[B]_{\mathtt{p}}$ configuration reached on $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$, B is in state t. Because $s_0 \xrightarrow{u}{}^* s \in \delta_A$, and all non-$\mathtt{p}$ state machines are identical from $\mathscr{C}(\mathbf{G})[B]_{\mathtt{p}}$ to $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$, it is clear that $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$ is also a trace of $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$ and in the CSM configuration reached on $\mathtt{split}(\mathtt{trace}(\alpha \cdot G))$, A is in state s. □

Having found our witness well-behaved context $\mathscr{C}(\mathbf{G})[\cdot]_{\mathtt{p}}$, established Lemma 5.16 to replace Lemma 5.8, and observed that the violation of each condition in C_2 likewise yields a state with a non-empty decoration set with respect to B, completeness then amounts to showing the existence of a $w \in \Sigma_{async}^*$ such that w refutes subprotocol fidelity, language inclusion, or deadlock freedom. Recall that the proofs for the necessity of *Send Preservation*, *Receive Exhaustivity* and *Final State Validity* in the case where

B is the subset construction constructed a trace that refuted either subprotocol fidelity or deadlock freedom. These two properties are identical across both formulations of the problem, and therefore the construction can be wholly reused to show the necessity of *Send Subtype Preservation*, *Receive Subtype Exhaustivity* and *Final State Subtype Validity*.

Lemma 5.17. *If* $\mathcal{A}[A]_{\mathtt{p}}$ *violates* Send Decoration Subtype Validity *or* Receive Decoration Subtype Validity*, then it does not hold that for all well-behaved contexts* $\mathcal{A}[\cdot]_{\mathtt{p}}$*,* $\mathcal{A}[A]_{\mathtt{p}}$ *refines* $\mathcal{A}[B]_{\mathtt{p}}$*.*

The proofs for the necessity of *Send Decoration Validity* and *Receive Decoration Validity*, on the other hand, construct a word that is a trace in $\mathcal{A}[A]_{\mathtt{p}}$ but not a trace in $\mathscr{C}(\mathbf{G})[A]_{\mathtt{p}}$. In the general case, we can show that the same construction is a trace in $\mathcal{A}[A]_{\mathtt{p}}$ but not a trace in $\mathcal{A}[B]_{\mathtt{p}}$. We omit the proofs to avoid redundancy.

Lemma 5.18. *If* $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *violates* Send Subtype Preservation*,* Receive Subtype Exhaustivity*, or* Final State Subtype Validity*, then it does not hold that for all well-behaved contexts* $\mathcal{A}[\cdot]_{\mathtt{p}}$*,* $\mathcal{A}[A]_{\mathtt{p}}$ *refines* $\mathcal{A}[B]_{\mathtt{p}}$*.*

6 Complexity Analysis

We complete our discussion with a complexity analysis of the two considered problems, building on the characterizations established in Theorem 4.1 and Theorem 5.4.

For the *Protocol Verification* problem, let m be the size of $\mathcal{A}$ and n the size of $\mathbf{G}$. Moreover, let $A_{\mathtt{p}}$ be the local implementation of some role $\mathtt{p}$ in $\mathcal{A}$. Observe that the sets $d_{\mathbf{G}}(s)$ for each state s of $A_{\mathtt{p}}$ as well as the sets $M^{\mathtt{p}}_{(G'\ldots)}$ for each subterm G' of $\mathbf{G}$ are at most of size n. It is then easy to see that C_1 can be checked in time polynomial in n and m, provided that the sets $d_{\mathbf{G}}(s)$ and $M^{\mathtt{p}}_{(G'\ldots)}$ are also computable in polynomial time.

To see this for the sets $M^{\mathtt{p}}_{(G'\ldots)}$, observe that the definition expands each occurrence of a recursion variable in $\mathbf{G}$ at most once. So the traversal takes time $O(n^2)$. For each traversed event $\mathtt{p}\to\mathtt{q}:m$ in $\mathbf{G}$, we need to perform a constant number of lookup, insertion, and deletion operations on a set of size at most n, which takes time $O(\log n)$. The time for computing $M^{\mathtt{p}}_{(G'\ldots)}$ is thus in $O(n^2 \log n)$.

Similarly, observe that the function $d_{\mathbf{G}}$ can be computed for the local implementation of each role $A_{\mathtt{p}} \in \mathcal{P}$ using a simple fixpoint loop. Each set $d_{\mathbf{G}}(s)$ can be represented as a bit vector of size n, making all set operations constant time. The loop inserts at most n subterms of $\mathbf{G}$ into each $d_{\mathbf{G}}(s)$, which takes time $O(mn)$ for all insertions. Moreover, for each G inserted into a set $d_{\mathbf{G}}(s)$ and each transition $s \xrightarrow{x} s'$ in $A_{\mathtt{p}}$, we need to compute the set $\{G' \mid G \xrightarrow{x}{}^{*} G' \in \delta_{\downarrow}\}$ which is then added to $d_{\mathbf{G}}(s')$. Computing these sets takes time $O(mn)$ for each G and s.

Following analogous reasoning, we can also establish that C_2 is checkable in polynomial time.

Theorem 6.1. *The* Protocol Verification *and* Protocol Refinement *problems are decidable in polynomial time.*

7 Related Work

Session types were first introduced in binary form by Honda in 1993 [29]. Binary session types describe interactions between two participants, and communication safety of binary sessions amounts to channel duality. Binary session types were generalized to multiparty session types – describing interactions between more than two participants – by Honda, Yoshida and Carbone in 2008 [31], and the corresponding notion of safety was generalized from duality to multiparty consistency. Binary session types were inspired by and enjoy a close connection to linear logic [11, 28, 50]. Horne generalizes this connection to multiparty session types and non-commutative extensions of linear logic [32]. The connection between multiparty session types and logic is also explored in [10, 12, 13]. MSTs have since been extensively studied and widely adopted in practical programming languages; we refer the reader to [19] for a comprehensive survey.

Session type syntax. Session type frameworks have enjoyed various extensions since their inception. In particular, the choice operator for both global and local types has received considerable attention over the years. MSTs were originally introduced as global types, with a *directed* choice operator that restricted a sender to sending different messages to the same recipient. [15] and [40] relax this restriction to *sender-driven choice*, which allows a sender to send different messages to different recipients, and increases the expressivity of global types. Our paper targets global types with sender-driven choice. For local types, a direct comparison can be drawn to the π-calculus, for which *mixed choice* was shown to be strictly more expressive than *separate choice* [43]. Mixed choices allow both send and receive actions, whereas separate choices consist purely of either sends or receives. [38] showed that any global type with sender-driven choice can be implemented by a CSM with only separate choice. Mixed choice for binary local types was investigated in [14], although [44] later showed that this variant falls short of the full expressive power of mixed choice π-calculus, and instead can only express separate choice π-calculus. Other communication primitives have also been studied, such as channel delegation [17, 30, 31], dependent predicates [48, 49], parametrization [18, 22] and data refinement [51].

Session type semantics. MSTs were introduced in [31] with a process algebra semantics. The connection to CSMs was established in [21], which defines a class of CSMs whose state machines can be represented as local types, called *Communicating Session Automata* (CSA). CSAs inherit from the local types they represent restrictions on choice discussed above, "tree-like" restrictions on the structure (see [47] for a characterization), and restrictions on outgoing transitions from final states. The CSM implementation model in our work assumes none of the above restrictions, and is thus true to its name.

Session subtyping. Session subtyping was first introduced by [25] in the context of the π-calculus, which was in turn inspired by Pierce and Sangiorgi's work on subtyping for channel endpoints [45]. The session types literature distinguishes between two notions of subtyping based on the network assumptions of the framework: *synchronous* and *asynchronous subtyping*. Both notions respect Liskov and Wing's substitution principle [39], but differ in the guarantees provided. We discuss each in turn.

Synchronous subtyping follows the notions of covariance and contravariance introduced by [25], and checks that a subtype contains fewer sends and more receives than its supertype. For binary synchronous session types, Lange and Yoshida [34] show that subtyping can be decided in quadratic time via model checking of a characteristic formulae in the modal μ-calculus. For multiparty synchronous session types, Ghilezan et al. [26] present a precise subtyping relation that is universally quantified over all contexts, and restricts the local type syntax to directed choice. As mentioned in §1, [26], their subtyping relation is incomplete when generalized to asynchronous multiparty sessions with directed choice. As discussed in §2, their subtyping relation is further incomplete when generalized to asynchronous multiparty sessions with mixed choice, due to the "peculiarity [...] that, apart from a pair of inactive session types, only inputs and outputs from/to a same participant can be related" [26]. The complexity of the subtyping relation in [26] is not mentioned.

Unlike subtyping relations for synchronous sessions which preserve language inclusion, subtyping relations for asynchronous sessions instead focus on deadlock-free optimizations that permute roles' local order of send and receive actions, also called *asynchronous message reordering*, or AMR [20]. First proposed for binary sessions by Mostrous and Yoshida [41], and for multiparty sessions by Mostrous et al. [42], this notion of subtyping does not satisfy subprotocol fidelity in general; indeed, in some cases, the set of behaviors recognized by a supertype is entirely disjoint from that of its subtype [5]. Asynchronous subtyping was shown to be undecidable for both binary and multiparty session types [6, 35]. Existing works are thus either restricted to binary protocols [1,5,6,35], prohibit non-deterministic choice involving multiple receivers [7,27], or make strong fairness assumptions on the network [7].

The connection between session subtyping and behavioral contract refinement has been studied only in the context of binary session types, and is thus out of scope of our work. We refer the reader to [26] for a survey.

Acknowledgements The authors thank Damien Zufferey for discussions and feedback. This work is funded in parts by the National Science Foundation under grant CCF-2304758. Felix Stutz was supported by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248—CPEC.

References

1. Bacchiani, L., Bravetti, M., Lange, J., Zavattaro, G.: A session subtyping tool. In: Damiani, F., Dardha, O. (eds.) Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12717, pp. 90–105. Springer (2021). `https://doi.org/10.1007/978-3-030-78142-2_6`
2. Barbanera, F., De'Liguoro, U.: Sub-behaviour relations for session-based client/server systems. Mathematical Structures in Computer Science **25**(6), 1339–1381 (2015). `https://doi.org/10.1017/S096012951400005X`

3. Bernardi, G.T., Hennessy, M.: Modelling session types using contracts. Math. Struct. Comput. Sci. **26**(3), 510–560 (2016). https://doi.org/10.1017/S0960129514000243
4. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
5. Bravetti, M., Carbone, M., Lange, J., Yoshida, N., Zavattaro, G.: A sound algorithm for asynchronous session subtyping and its implementation. Log. Methods Comput. Sci. **17**(1) (2021), https://lmcs.episciences.org/7238
6. Bravetti, M., Carbone, M., Zavattaro, G.: On the boundary between decidability and undecidability of asynchronous session subtyping. Theor. Comput. Sci. **722**, 19–51 (2018). https://doi.org/10.1016/j.tcs.2018.02.010
7. Bravetti, M., Lange, J., Zavattaro, G.: Fair refinement for asynchronous session types. In: Kiefer, S., Tasson, C. (eds.) Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12650, pp. 144–163. Springer (2021). https://doi.org/10.1007/978-3-030-71995-1_8
8. Bravetti, M., Zavattaro, G.: Relating session types and behavioural contracts: The asynchronous case. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 29–47. Springer International Publishing, Cham (2019)
9. Bravetti, M., Zavattaro, G.: Asynchronous session subtyping as communicating automata refinement. Softw. Syst. Model. **20**(2), 311–333 (apr 2021). https://doi.org/10.1007/s10270-020-00838-x
10. Caires, L., Pérez, J.A.: Multiparty session types within a canonical binary theory, and beyond. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9688, pp. 74–95. Springer (2016). https://doi.org/10.1007/978-3-319-39570-8_6
11. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Math. Struct. Comput. Sci. **26**(3), 367–423 (2016). https://doi.org/10.1017/S0960129514000218
12. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: A logical explanation of multiparty session types. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.33
13. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. Acta Informatica **54**(3), 243–269 (2017). https://doi.org/10.1007/s00236-016-0285-y
14. Casal, F., Mordido, A., Vasconcelos, V.T.: Mixed sessions. Theor. Comput. Sci. **897**, 23–48 (2022). https://doi.org/10.1016/j.tcs.2021.08.005
15. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party session. Log. Methods Comput. Sci. **8**(1) (2012). https://doi.org/10.2168/LMCS-8(1:24)2012

16. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. **31**(5), 19:1–19:61 (2009). `https://doi.org/10.1145/1538917.1538920`
17. Castellani, I., Dezani-Ciancaglini, M., Giannini, P., Horne, R.: Global types with internal delegation. Theor. Comput. Sci. **807**, 128–153 (2020). `https://doi.org/10.1016/j.tcs.2019.09.027`
18. Charalambides, M., Dinges, P., Agha, G.A.: Parameterized, concurrent session types for asynchronous multi-actor interactions. Sci. Comput. Program. **115-116**, 100–126 (2016). `https://doi.org/10.1016/j.scico.2015.10.006`
19. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures. Lecture Notes in Computer Science, vol. 9104, pp. 146–178. Springer (2015). `https://doi.org/10.1007/978-3-319-18941-3_4`
20. Cutner, Z., Yoshida, N., Vassor, M.: Deadlock-free asynchronous message reordering in rust with multiparty session types. In: Lee, J., Agrawal, K., Spear, M.F. (eds.) PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022. pp. 246–261. ACM (2022). `https://doi.org/10.1145/3503221.3508404`
21. Deniélou, P., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7211, pp. 194–213. Springer (2012). `https://doi.org/10.1007/978-3-642-28869-2_10`
22. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Log. Methods Comput. Sci. **8**(4) (2012). `https://doi.org/10.2168/LMCS-8(4:6)2012`
23. Ellul, K., Krawetz, B., Shallit, J.O., Wang, M.: Regular expressions: New results and open problems. J. Autom. Lang. Comb. **10**(4), 407–437 (2005). `https://doi.org/10.25596/jalc-2005-407`
24. Esparza, J., Nielsen, M.: Decidability issues for petri nets - a survey. J. Inf. Process. Cybern. **30**(3), 143–160 (1994)
25. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2-3), 191–225 (2005). `https://doi.org/10.1007/s00236-005-0177-z`
26. Ghilezan, S., Jakšić, S., Pantović, J., Scalas, A., Yoshida, N.: Precise subtyping for synchronous multiparty sessions. Journal of Logical and Algebraic Methods in Programming **104**, 127–173 (2019). `https://doi.org/https://doi.org/10.1016/j.jlamp.2018.12.002`
27. Ghilezan, S., Pantovic, J., Prokic, I., Scalas, A., Yoshida, N.: Precise subtyping for asynchronous multiparty sessions. Proc. ACM Program. Lang. **5**(POPL), 1–28 (2021). `https://doi.org/10.1145/3434297`
28. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987). `https://doi.org/10.1016/0304-3975(87)90045-4`

29. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35
30. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
31. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472
32. Horne, R.: Session subtyping and multiparty compatibility using circular sequents. In: Konnov, I., Kovács, L. (eds.) 31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference). LIPIcs, vol. 171, pp. 12:1–12:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.12
33. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563
34. Lange, J., Yoshida, N.: Characteristic formulae for session types. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 833–850. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_52
35. Lange, J., Yoshida, N.: On the undecidability of asynchronous session subtyping. In: Esparza, J., Murawski, A.S. (eds.) Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10203, pp. 441–457 (2017). https://doi.org/10.1007/978-3-662-54458-7_26
36. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 97–117. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_6
37. Li, E., Stutz, F., Wies, T.: Deciding subtyping for asynchronous multiparty sessions. CoRR **abs/2401.16395** (2024). https://doi.org/10.48550/arXiv.2401.16395

38. Li, E., Stutz, F., Wies, T., Zufferey, D.: Complete multiparty session type projection with automata. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 350–373. Springer Nature Switzerland, Cham (2023)
39. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994). https://doi.org/10.1145/197320.197383
40. Majumdar, R., Mukund, M., Stutz, F., Zufferey, D.: Generalising projection in asynchronous multiparty session types. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 35:1–35:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.35
41. Mostrous, D., Yoshida, N.: Session-based communication optimisation for higher-order mobile processes. In: Curien, P. (ed.) Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5608, pp. 203–218. Springer (2009). https://doi.org/10.1007/978-3-642-02273-9_16
42. Mostrous, D., Yoshida, N., Honda, K.: Global principal typing in partially commutative asynchronous sessions. In: Castagna, G. (ed.) Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5502, pp. 316–332. Springer (2009). https://doi.org/10.1007/978-3-642-00590-9_23
43. Palamidessi, C.: Comparing the expressive power of the synchronous and asynchronous pi-calculi. Math. Struct. Comput. Sci. **13**(5), 685–719 (2003). https://doi.org/10.1017/S0960129503004043
44. Peters, K., Yoshida, N.: On the expressiveness of mixed choice sessions. In: Castiglioni, V., Mezzina, C.A. (eds.) Proceedings Combined 29th International Workshop on Expressiveness in Concurrency and 19th Workshop on Structural Operational Semantics, EXPRESS/SOS 2022, and 19th Workshop on Structural Operational Semantics Warsaw, Poland, 12th September 2022. EPTCS, vol. 368, pp. 113–130 (2022). https://doi.org/10.4204/EPTCS.368.7
45. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Math. Struct. Comput. Sci. **6**(5), 409–453 (1996). https://doi.org/10.1017/s096012950007002x
46. Sipser, M.: Introduction to the theory of computation. PWS Publishing Company (1997)
47. Stutz, F.: Asynchronous multiparty session type implementability is decidable - lessons learned from message sequence charts. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 32:1–32:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.ECOOP.2023.32
48. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: Schneider-Kamp, P., Hanus, M. (eds.) Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark. pp. 161–172. ACM (2011). https://doi.org/10.1145/2003476.2003499
49. Toninho, B., Caires, L., Pfenning, F.: A decade of dependent session types. In: 23rd International Symposium on Principles and Practice of Declarative Programming. PPDP 2021,

Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3479394.3479398

50. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2-3), 384–418 (2014). https://doi.org/10.1017/S095679681400001X
51. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proceedings of the ACM on Programming Languages **4**, 1–30 (11 2020). https://doi.org/10.1145/3428216

The Session Abstract Machine

Luís Caires[1] and Bernardo Toninho[2](✉)

[1] Técnico Lisboa / INESC-ID, Lisbon, Portugal
`luis.caires@tecnico.ulisboa.pt`
[2] NOVA FCT / NOVA LINCS, Costa da Caparica, Portugal
`btoninho@fct.unl.pt`

Abstract. We build on a fine-grained analysis of session-based interaction as provided by the linear logic typing disciplines to introduce the SAM, an abstract machine for mechanically executing session-typed processes. A remarkable feature of the SAM's design is its ability to naturally segregate and coordinate sequential with concurrent session behaviours. In particular, implicitly sequential parts of session programs may be efficiently executed by deterministic sequential application of SAM transitions, amenable to compilation, and without concurrent synchronisation mechanisms. We provide an intuitive discussion of the SAM structure and its underlying design, and state and prove its correctness for executing programs in a session calculus corresponding to full classical linear logic CLL. We also discuss extensions and applications of the SAM to the execution of linear and session-based programming languages.

Keywords: Abstract machine · Session Types · Linear Logic

1 Introduction

In this work, we build on the linear logic based foundation for session types [13, 15, 72] to construct SAM, an abstract machine specially designed for executing session processes typed by (classical) linear logic CLL. Although motivated by the session type discipline, which originally emerged in the realm of concurrency and distribution [31, 33, 28, 34], a basic motivation for designing the SAM was to provide an efficient deterministic execution model for the implicitly sequential session-typed program idioms that often proliferate in concurrent session-based programming. It is well-known that in a world of fine-grained concurrency, building on many process-based encodings of concepts such as (abstract) data types, functions, continuations, and effects [49, 70, 65, 66, 10, 68, 54], large parts of the code turn out to be inherently sequential, further justifying the foundational and practical relevance of our results. A remarkable feature of the SAM's design is therefore its potential to efficiently coordinate sequential with full-fledged concurrent behaviours in session-based programming.

Leveraging early work relating linear logic with the semantics of linear and concurrent computation [1, 6, 2], the proposition-as-types (PaT) interpretation [73] of linear logic proofs as a form of well-behaved session-typed nominal calculus has motivated many developments since its inception [12, 5, 68, 67]. We believe that, much how the λ-calculus is deemed a canonical typed model for functional (sequential) computation with pure values, the session calculus can be accepted as a fairly canonical typed model for stateful concurrent computation with linear resources, well-rooted in the trunk of "classical" Type Theory. The PaT interpretation of session processes also establishes a bridge between more classical theories of computation and process algebra via logic. It also reinstates Robin Milner's view of computation as interaction [48], "data-as-processes" [49] and "functions-as-processes" [47], now in the setting of a tightly typed world, based on linear logic, where types may statically ensure key properties like deadlock-freedom, termination, and correct resource usage in stateful programs. Session calculi are motivating novel programming language design, bringing up new insights on typeful programming [18] with linear and behavioral types, e.g., [24, 61, 20, 5]. Most systems of typed session calculi have been formulated in process algebraic form [31, 33, 28], or on top of concurrent λ-calculi with an extra layer of communication channels (e.g., [29]), logically inspired systems such as the those discussed in this paper (e.g., [13, 15, 72, 23, 39, 59, 27, 61]) are defined by a logical proof / type system where proof rules are seen as witnesses for the typing of process terms, proofs are read as processes, structural equivalence is proof conversion and computation corresponds to cut reduction. These formulations provide a fundamental semantic foundation to study the model's expressiveness and meta-theory, but of course do not directly support the concrete implementation of programming languages based on them.

Although several programming language implementations of nominal calculi based languages have been proposed for some time (e.g. [57]), with some introducing abstract machines as the underlying technology (e.g., [69, 46]), we are not aware of any prior design proposal for an abstract machine for reducing session processes exploiting deep properties of a source session calculus, as e.g., the CAM [21] the LAM [41], or the KM [40], which also explore the Curry-Howard correspondences, may reclaim to be, respectively for call-by-value cartesian-closed structures, linear logic, and the call-by-name λ-calculus.

The SAM reduction strategy explores a form of "asynchronous" interaction that essentially expresses that, for processes typed by the logical discipline, sessions are always pairwise causally independent, in the sense that immediate communication on some session is never blocked by communication on a different session. This property is captured syntactically by prefix commutation equations, valid commuting conversions in the underlying logic: adding equations for such laws explicitly to process structural congruence keeps observational equivalence of CLL processes untouched [53]. Combined with insights related to focalisation and polarisation in linear logic [4, 56, 44], we realize that all communication in any session may be operationally structured as the exchange of bundles of positive actions from sender to receiver, where the roles sender/receiver flip whenever the session type swaps polarity. Communication may then be mediated by message buffers, first filled up by the sender ("write-biased" scheduling), and at a later time emptied by the receiver. Building on these observations and on key properties of linear logic proofs leveraged in well-known purely structural proofs of progress [13, 15, 61], we identify a sequential and deterministic reduction strategy for CLL typed processes, based on a form of co-routining where continuations are associated to session queues, and "context switching" occurs whenever polarity flips. That such strategy works at all, preserving all the required correctness properties of the CLL language does not seem immediately obvious, given that each processes may sequentially perform multiple actions on many different sessions, meaning that multiple context switches must be interleaved. The bulk of our paper is then devoted to establishing all such properties in a precise technical sense. We believe that the SAM may provide a principled foundation for safe execution environments for programming languages combining functional, imperative and concurrent idioms based on session and linear types, as witnessed in practice for Rust [37], (Linear) Haskell [45, 8, 38], Move [9], and in research languages [60, 36, 24]. To further substantiate these views we have developed an implementation of the SAM, integrated in a language for realistic session-based shared-state programs [17].

Outline and Contributions. In Section 2 we briefly review the session-typed calculus CLL, which exactly corresponds to (classical) Linear Logic with mix. In Section 3 we discuss the motivation and design principles of the core SAM, gradually presenting its structure for the language fragment corresponding to session types without the exponentials, which will be introduced later. Even if the core SAM structure and transition rules are fairly simple, the proofs of correctness are more technically involved, and require progressive build up. Therefore, we first bridge between CLL and SAM via a intermediate logical language CLLB, introducing explicit queues in cuts, presented in Section 4. We show preservation (Theorem 4.1) and progress (Theorem 4.2) for CLLB, and prove that there is two way simulation between CLLB and CLL via a strong operational correspondence (Theorem 4.3). Given this correspondence, in Section 5 we state and prove the adequacy of the SAM for executing CLL processes, showing soundness wrt. CLLB (Theorem 5.1) and CLL (Theorem 5.2), and progress / deadlock absence (Theorem 5.3). In Section 6 modularly extend the previous results to the exponentials and mix, and revise the core SAM by introducing explicit environments, stating the associated adequacy results (Theorem 6.1 and Theorem 6.2). We also discuss how to accommodate concurrency, and other extensions in the SAM. We conclude by a discussion of related work and additional remarks. Additional definitions and proofs can be found in the companion technical report [16].

2 Background on CLL, the core language and type system

We start by revisiting the language and type system of CLL, and its operational semantics. The system is based on a PaT interpretation of classical linear logic (we follow the presentations of [15, 11, 60]).

Definition 2.1 (Types). *Types A, B are defined by*

$$A, B ::= \mathbf{1} \mid \bot \mid A \parr B \mid A \otimes B \mid \&_{\ell \in L} A_\ell \mid \oplus_{\ell \in L} A_\ell \mid !A \mid ?A$$

Types comprise of the units $(\mathbf{1}, \bot)$, multiplicatives $(\otimes, \parr)$, additives $(\oplus_{\ell \in L} A_\ell, \&_{\ell \in L} A_\ell)$ and exponentials $(!, ?)$. We adopt here a labeled version of the additives, where the linear logic sum type $A_{\#\mathsf{inl}} \oplus A_{\#\mathsf{inr}}$ is defined by $\oplus_{\ell \in \{\#\mathsf{inl}, \#\mathsf{inr}\}} A_\ell$. The *positive types* are $\mathbf{1}$, $\otimes$, $\oplus$, and $!$, while the *negative types* are $\bot$, $\parr$, $\&$ and $?$. We abbreviate $\overline{A} \parr B$ by $A \multimap B$. We write A^+ (resp. A^-) to assert that A is a positive (resp. negative) type. Type *duality* $\overline{A}$ corresponds to negation:

$$\overline{\mathbf{1}} = \bot \qquad \overline{A \otimes B} = \overline{A} \parr \overline{B} \qquad \overline{\oplus_{\ell \in L} A_\ell} = \&_{\ell \in L} \overline{A_\ell} \qquad \overline{!A} = ?\overline{B}$$

Duality captures the symmetry of behaviour in binary process interaction, as manifest in the cut rule.

Definition 2.2 (Processes). *The syntax of processes P, Q is given by:*

$$\begin{array}{rl} P, Q ::= & \mathsf{0} \mid P \mathbin{||} Q \mid \mathsf{fwd}\ x\ y \mid \mathsf{cut}\ \{P\ |x{:}A|\ Q\} \mid \mathsf{close}\ x \mid \mathsf{wait}\ x; P \\ \mid & \mathsf{case}\ x\ \{|\#\ell \in L{:}P_\ell\} \mid \#\mathsf{inr}\ x; P \mid \mathsf{send}\ x(y.P); Q \mid \mathsf{recv}\ x(z); P \\ \mid & !x(y); P \mid ?x; P \mid \mathsf{cut!}\ \{y.P\ |!x : A|\ Q\} \mid \mathsf{call}\ x(z); Q \end{array}$$

Typing judgements have the form $P \vdash \Delta; \Gamma$, where P is a process and the *typing context* $\Delta; \Gamma$ is dyadic [4, 7, 55, 13]: both Δ and Γ assign types to names, the context Δ is handled linearly (no implicit contraction or weakening) while the exponential context Γ is unrestricted. The type system exactly corresponds, via a propositions-as-types correspondence, to the canonical proof system of Classical Linear Logic with Mix. When a cut type annotation is easily inferred, we may omit it and write $\mathsf{cut}\ \{P\ |x|\ Q\}$. The typing rules of CLL are given in Fig. 1.

The process $\mathsf{0}$ denotes the inactive process, typed in the empty linear context (rule [T$\mathsf{0}$]). $P \mathbin{||} Q$ denotes independent parallel composition of processes P and Q (rule [Tmix]), whereas $\mathsf{cut}\ \{P\ |x{:}A|\ Q\}$ denotes interfering parallel composition of P and Q, where P and Q share exactly one channel name x, typed as A in P and $\overline{A}$ in Q (rule [Tcut]). The construct $\mathsf{fwd}\ x\ y$ captures forwarding between dually typed names x and y (rule [Tfwd]), which operationally consists in (globally) renaming x for y.

Processes $\mathsf{close}\ x$ and $\mathsf{wait}\ x; P$ denote session termination and the dual action of waiting for session termination, respectively (rules [T$\mathbf{1}$] and [T$\bot$]). The constructs $\mathsf{case}\ x\ \{|\#\ell \in L{:}P_\ell\}$ and $\#\mathsf{l}\ x; P$ denote label input and output, respectively, where the input construct pattern matches on the received label to select the process continuation that is to run. Process $\mathsf{send}\ x(y.P_1); P_2$ and $\mathsf{recv}\ x(z); Q$ codify the output of (fresh) name y on channel x and the corresponding input action, where the received name will be substituted for z in Q (rules [T$\otimes$] and [T$\parr$]). Typing ensures that the names used in P_1 and P_2 are disjoint.

Processes $!x(y); P$, $?x; Q$ and $\mathsf{call}\ x(z); Q$ embody replicated servers and client processes. Process $!x(y); P$ consists of a process that waits for inputs on x, spawning a replica of P (depending on no linear sessions – rule [T!]). Process $?x; Q$ and $\mathsf{call}\ x(z); Q$ allow for replicated servers to be activated and subsequently used as (fresh) linear sessions (rules [T?] and [Tcall]). Composition of exponentials is achieved by the $\mathsf{cut!}\ \{y.P\ |!x : A|\ Q\}$ process, where P cannot depend on linear sessions and so may be safely replicated.

We call *action* any process that is either a forwarder or realizes an introduction rule, and denote by $\mathcal{A}$ the set of all actions, by $\mathcal{A}(x)$ the set of action with subject x (the subject of an action is the channel name in which it interacts [49]). An action is deemed *positive* (resp. *negative*) if its associated type is positive (resp. negative) in the sense of *focusing*. The set of positive (resp.

$$\frac{}{\mathsf{0} \vdash \emptyset; \Gamma}\,[\mathrm{T}\mathsf{0}] \qquad \frac{P \vdash \Delta'; \Gamma \quad Q \vdash \Delta; \Gamma}{P \parallel Q \vdash \Delta', \Delta; \Gamma}\,[\mathrm{Tmix}]$$

$$\frac{}{\mathsf{fwd}\ x\ y \vdash x : \overline{A}, y : A; \Gamma}\,[\mathrm{Tfwd}] \qquad \frac{P \vdash \Delta', x : A; \Gamma \quad Q \vdash \Delta, x : \overline{A}; \Gamma}{\mathsf{cut}\ \{P \mid x : A \mid Q\} \vdash \Delta', \Delta; \Gamma}\,[\mathrm{Tcut}]$$

$$\frac{}{\mathsf{close}\ x \vdash x : \mathbf{1}; \Gamma}\,[\mathrm{T}\mathbf{1}] \qquad \frac{Q \vdash \Delta; \Gamma}{\mathsf{wait}\ x; Q \vdash \Delta, x : \bot; \Gamma}\,[\mathrm{T}\bot]$$

$$\frac{P_\ell \vdash \Delta, x : A_\ell; \Gamma \quad (\mathit{all}\ \ell \in L)}{\mathsf{case}\ x\ \{|\#\ell \in L{:}P_\ell\} \vdash \Delta, x : \&_{\ell \in L} A_\ell; \Gamma}\,[\mathrm{T}\&] \qquad \frac{Q \vdash \Delta', x : A_{\#\mathsf{l}}; \Gamma \quad \#\mathsf{l} \in L}{\#\mathsf{l}\ x; Q \vdash \Delta', x : \oplus_{\ell \in L} A_\ell; \Gamma}\,[\mathrm{T}\oplus]$$

$$\frac{P_1 \vdash \Delta_1, y : A; \Gamma \quad P_2 \vdash \Delta_2, x : B; \Gamma}{\mathsf{send}\ x(y.P_1); P_2 \vdash \Delta_1, \Delta_2, x : A \otimes B; \Gamma}\,[\mathrm{T}\otimes]$$

$$\frac{Q \vdash \Delta, z : A, x : B; \Gamma}{\mathsf{recv}\ x(z); Q \vdash \Delta, x : A \parr B; \Gamma}\,[\mathrm{T}\parr]$$

$$\frac{P \vdash y : A; \Gamma}{!x(y); P \vdash x : !A; \Gamma}\,[\mathrm{T!}] \qquad \frac{Q \vdash \Delta; \Gamma, x : A}{?x; Q \vdash \Delta, x : ?A; \Gamma}\,[\mathrm{T?}]$$

$$\frac{P \vdash y : A; \Gamma \quad Q \vdash \Delta; \Gamma, x : \overline{A}}{\mathsf{cut!}\ \{y.P \mid !x : A \mid Q\} \vdash \Delta; \Gamma}\,[\mathrm{Tcut!}] \qquad \frac{Q \vdash \Delta, z : A; \Gamma, x : A}{\mathsf{call}\ x(z); Q \vdash \Delta; \Gamma, x : A}\,[\mathrm{Tcall}]$$

Fig. 1: Typing Rules of CLL.

negative) actions is denoted by $\mathcal{A}^+$ (resp. $\mathcal{A}^-$). We sometimes use, e.g., $\mathcal{A}$ or $\mathcal{A}^+(x)$ to denote a process in the set. The CLL operational semantics is given by a *structural congruence* relation $\equiv$ that captures static identities on processes, corresponding to commuting conversions in the logic, and a *reduction* relation $\rightarrow$ that captures process interaction, and corresponds to cut-elimination steps.

Definition 2.3 **($P \equiv Q$).** *Structural congruence $\equiv$ is the least congruence on processes closed under α-conversion and the $\equiv$-rules in Fig. 2.*

The definition of $\equiv$ reflects expected static laws, along the lines of the structural congruences / conversions in [13, 71]. The binary operators forwarder, cut, and mix are commutative. The set of processes modulo $\equiv$ is a commutative monoid with operation the parallel composition $(- \parallel -)$ and identity given by inaction $\mathsf{0}$ ([par]). Any static constructs commute, as expressed by the laws [CM]-[C!sC!]. The unrestricted cut distributes over all the static constructs by law [C*], where $- \mid * \mid -$ stands for either a mix, linear or unrestricted cut. The laws [C+*] and [C+] denote sound proof equivalences in linear logic and bring explicit the independence of linear actions (noted $a(x)$), in different sessions x [53]. These conversions are not required to obtain deadlock freedom. However, they are necessary for full cut elimination (e.g., see [71]), and expose more redexes, thus more non-determinism in the choice of possible reductions. Perhaps surprisingly,

$$\mathsf{fwd}\ x\ y\ \equiv \mathsf{fwd}\ y\ x \qquad [\text{fwd}]$$
$$\mathsf{cut}\ \{P\ |x:A|\ Q\}\ \equiv \mathsf{cut}\ \{Q\ |x:\overline{A}|\ P\} \qquad [\text{com}]$$
$$P \mathbin{||} \mathsf{0}\ \equiv P \quad P \mathbin{||} Q \equiv Q \mathbin{||} P \quad P \mathbin{||} (Q \mathbin{||} R) \equiv (P \mathbin{||} Q) \mathbin{||} R \qquad [\text{par}]$$
$$\mathsf{cut}\ \{P\ |x|\ (Q \mathbin{||} R)\} \equiv (\mathsf{cut}\ \{P\ |x|\ Q\}) \mathbin{||} R \qquad [\text{CM}]$$
$$\mathsf{cut}\ \{P\ |x|\ (\mathsf{cut}\ \{Q\ |y|\ R\})\} \equiv \mathsf{cut}\ \{(\mathsf{cut}\ \{P\ |x|\ Q\})\ |y|\ R\} \qquad [\text{CC}]$$
$$\mathsf{cut}\ \{P\ |z|\ (\mathsf{cut!}\ \{y.Q\ |!x|\ R\})\} \equiv \mathsf{cut!}\ \{y.Q\ |!x|\ (\mathsf{cut}\ \{P\ |z|\ R\})\} \qquad [\text{CC!}]$$
$$\mathsf{cut!}\ \{y.Q\ |!x|\ (P \mathbin{||} R)\} \equiv P \mathbin{||} (\mathsf{cut!}\ \{y.Q\ |!x|\ R\}) \qquad [\text{C!M}]$$
$$\mathsf{cut!}\ \{y.P\ |!x|\ (\mathsf{cut!}\ \{w.Q\ |!z|\ R\})\} \equiv \mathsf{cut!}\ \{w.Q\ |!z|\ (\mathsf{cut!}\ \{y.P\ |!x|\ R\})\} \qquad [\text{C!C!}]$$
$$\mathsf{cut!}\ \{y.P\ |!x|\ (Q\ |*|\ R)\} \equiv \mathsf{cut!}\ \{y.P\ |!x|\ Q\}\ |*|\ \mathsf{cut!}\ \{y.P\ |!x|\ R\} \qquad [\text{C!*}]$$
$$a(x); Q\ |*|\ R\ \equiv a(x); (Q\ |*|\ R) \qquad [\text{C+*}]$$
$$a_1(x); a_2(y); P\ \equiv a_2(y); a_1(x); P \qquad [\text{C}i]$$

Provisos: in [CM] $x \in \mathsf{fn}(Q)$; in [CC] $x, y \in \mathsf{fn}(Q)$; in [CC!], [C!M] $x \notin \mathsf{fn}(P)$; in [C!C!], $x \notin \mathsf{fn}(Q)$ and $z \notin \mathsf{fn}(P)$. In [Ci], $x \neq y$ and $bn(a_1(x)) \cap bn(a_2(y)) = \emptyset$

Fig. 2: Structural congruence $P \equiv Q$.

$$\mathsf{cut}\ \{\mathsf{fwd}\ x\ y\ |y|\ P\} \to \{x/y\}P \qquad [\text{fwd}]$$
$$\mathsf{cut}\ \{\mathsf{close}\ x\ |x|\ \mathsf{wait}\ x; P\} \to P \qquad [\mathbf{1}\bot]$$
$$\mathsf{cut}\ \{\mathsf{send}\ x(y.P); Q\ |x|\ \mathsf{recv}\ x(z); R\} \to Q\ |x|\ (P\ |y|\ \{y/z\}R) \qquad [\otimes⅋]$$
$$\mathsf{cut}\ \{\mathsf{case}\ x\ \{|\#\ell \in L{:}P_{\#\ell}\}\ |x|\ \#\mathsf{l}\ x; R\} \to \mathsf{cut}\ \{P_{\#\mathsf{l}}\ |x|\ R\} \qquad [\&\oplus_l]$$
$$\mathsf{cut}\ \{!x(y); P\ |x|\ ?x; Q\} \to \mathsf{cut!}\ \{y.P\ |!x|\ Q\} \qquad [!?]$$
$$\mathsf{cut!}\ \{y.P\ |!x|\ \mathsf{call}\ x(z); Q\} \to \mathsf{cut}\ \{\{z/y\}P\ |z|\ (\mathsf{cut!}\ \{y.P\ |!x|\ Q\})\} \qquad [\text{call}]$$

Fig. 3: Reduction $P \to Q$.

this extra flexibility is important to allow the deterministic sequential evaluation strategy for CLL programs adopted by the SAM to be expressed.

Definition 2.4 (Reduction $\to$). *Reduction $\to$ is defined by the rules of Fig. 3.*

We denote by $\Rightarrow$ the reflexive-transitive closure of $\to$. Reduction includes the set of principal cut conversions, i.e. the redexes for each pair of interacting constructs. It is closed by structural congruence ([$\equiv$]), in rule [cong] we consider that $\mathcal{C}$ is a static context, i.e. a process context in which the single hole is covered only by the static constructs mix or cut. The forwarding behaviour is implemented by name substitution [fwd] [14]. All the other reductions act on a principal cut between two dual actions, and eliminate it on behalf of cuts involving their subprocesses. CLL satisfies basic safety properties [13] listed below, and also confluence, and termination [60, 61]. In particular we have:

Theorem 2.1 (Type Preservation). *Let $P \vdash \Delta; \Gamma$. (1) If $P \equiv Q$, then $Q \vdash \Delta; \Gamma$. (2) If $P \to Q$, then $Q \vdash \Delta; \Gamma$.*

A process P is *live* if and only if $P = \mathcal{C}[Q]$, for some static context $\mathcal{C}$ (the hole lies within the scope of static constructs mix and cut) and Q is an active process (a process with a topmost action prefix).

Theorem 2.2 (Progress). *Let $P \vdash \emptyset; \emptyset$ be live. Then $P \to Q$ for some Q.*

3 A Core Session Abstract Machine

In this section we develop the key insights that guide the construction of our session abstract machine (SAM) and introduce its operational rules in an incremental fashion. We omit the linear logic exponentials for the sake of clarity of presentation, postponing their discussion for Section 6.

One of the main observations that drives the design of the SAM is the nature of proof dynamics in (classical) linear logic, and thus of process execution dynamics in the CLL system of Section 2. The proof dynamics of linear logic are derived from the computational content of the cut elimination proof, which defines a proof simplification strategy that removes (all) instances of the cut rule from a proof. However, the strategy induced by cut elimination is *non-deterministic* insofar as multiple simplification steps may apply to a given proof. Transposing this observation to CLL and other related systems, we observe that their operational semantics is does not prescribe a rigid evaluation order for processes. For instance, in the process $\mathsf{cut}\ \{P\ |x|\ Q\}$, reduction is allowed in both P and Q. This is of course in line with reduction in process calculi (e.g., [49]). However, in logical-based systems this amounts to *don't care* non-determinism since, regardless of the evaluation order, confluence ensures that the same outcomes are produced (in opposition to *don't know* non-determinism which breaks confluence and is thus disallowed in purely logical systems). The design of the SAM arises from attempting to fix a purely sequential reduction strategy for CLL processes, such that only *one* process is allowed to execute at any given point in time, in the style of coroutines. To construct such a strategy, we forego the use of purely synchronous communication channels, which require a handshake between two concurrently executing processes, and so consider session channels as a kind of *buffered* communication medium (this idea has been explored in the context of linear logic interpretations of sessions in [25]), or queue, where one process can asynchronously write messages so that another may, subsequently, read. To ensure the correct directionality of communication, the queue has a write endpoint (on which a process may only write) and a read endpoint (along which only reads may be performed), such that at any given point in time a process can only hold one of two endpoints of a queue. Moreover, our design takes inspiration from insights related to polarisation and focusing in linear logic, grouping communication in sequences of positive (i.e. write) actions.

Allowing session channels to buffer message sequences, we may then model process execution by alternating between writer processes (that inject messages into the respective queues) and corresponding reader processes. Thus, the SAM

$$
\begin{array}{lll}
S & ::= (P, H) & \text{State} \\
H & ::= \mathit{SessionRef} \rightarrow \mathit{SessionRec} & \text{Heap} \\
R & ::= x\langle q, P\rangle y & \text{Session Record} \\
q & ::= \mathsf{nil} \mid \mathit{Val}@q & \text{Queue} \\
\mathit{Val} & ::= \checkmark & \text{Close token} \\
 & \quad\mid\ \#\mathsf{l} & \text{Choice label} \\
 & \quad\mid\ \mathsf{clos}(x, P) & \text{Process Closure}
\end{array}
$$

Fig. 4: core SAM Components

must maintain a *heap* that tracks the queue contents of each session (and its endpoints), as well as the suspended processes. The construction of the core of the SAM is given in Figure 4. An execution state is simply a pair consisting of *the* running process P and the heap H. For technical reasons that are made clear in Sections 4 and 5, the process language used in the SAM differs superficially from that of CLL, but for the purposes of this overview we will use CLL process syntax. Later we show the two languages are equivalent in a strong sense.

A heap is a mapping between session identifiers and *session records* of the form $x\langle q, Q\rangle y$, denoting a session with write endpoint x and read endpoint y, with queue contents q and a suspended process Q, holding one of the two endpoints. If Q holds the read endpoint then it is suspended waiting for the process holding the write endpoint to fill the queue with data for it to read. If Q holds the write endpoint, then Q has been suspended *after* filling the queue and is now waiting for the reader process on y to empty the queue.

We adopt the convention of placing the write endpoint on the left and the read endpoint on the right. In general, session records in the SAM support a form of coroutines through their contained processes, which are called on and returned from multiple times over the course of the execution of the machine. A queue can either be empty (nil) or holding a sequence of values. A value is either a close session token ($\checkmark$), identifying the last output on a session; a choice label $\#\mathsf{l}$ or a process closure $\mathsf{clos}(x, P)$, used to model session send and receive. We overload the @ notation to also denote concatenation of queues.

Cut. We begin by considering how to execute a cut of the form $\mathsf{cut}\ \{P \mid x : A^+ \mid Q\}$ where x is a positive type (in the sense of polarized logic [30]) in P. A positive type corresponds to a type denoting an *output* (or write) action, whereas a negative type denotes an *input* (or read) action. We maintain the invariant that in such a cut, P holds the write endpoint and Q the read endpoint. This means that the next action performed by P on the session will be to push some value onto the queue and, dually, the next action performed by Q on the session will be to read a value from the queue. In general, the holder of the write and read endpoint can change throughout execution.

Given the choice of either scheduling P or Q, we are effectively *forced* to schedule P *before* Q. Given that the cut introduces the (unique) session that is shared between the two processes, the only way for Q to exercise its read capability on the session successfully is to wait for P to have exercised (at least some of) its write capability. If we were to schedule Q before P, the process might attempt to read a value from an empty queue, resulting in a stuck state of the SAM. Thus, the SAM execution rule for cut is:

$$(\mathsf{cut}\ \{P\ |x : A^{+}|\ Q\}, H) \Rightarrow (P, H[x\langle \mathsf{nil}, \{y/x\}Q\rangle y]) \qquad [\text{SCut}]$$

The rule states that P is the process that is to be scheduled, adding the session record $x\langle \mathsf{nil}, Q\rangle y$ to the heap, which effectively suspends the execution of Q until P has exercised some of its write capabilities on the new session. Note that, in general, both P and Q can interact along many different sessions as both readers and writers before exercising any action on x (resp. y). However, they alone hold the freshly created endpoints x and y and so the next value sent along the session must come from P and Q is its intended receiver.

Channel Output. To execute an output of the form $\mathsf{send}\ x(z.R); Q$ in the SAM we simply lookup the session record for x and add to the queue a *process closure* containing R (which interacts along z), continuing with the execution of Q:

$$(\mathsf{send}\ x(z.R); Q, H[x\langle q, P\rangle y]) \Rightarrow (Q, H[x\langle q @ \mathsf{clos}(z, R), P\rangle y]) \qquad [\text{S}\otimes]$$

Note that the SAM *eagerly* continues to execute Q instead of switching to P, the holder of the read endpoint of the queue. This allows for the running process to perform all available writes before a context switch occurs.

Session Closure. Executing a close follows a similar spirit, but no continuation process exists and so execution switches to the process P holding the *read* endpoint y of the queue:

$$(\mathsf{close}\ x, H[x\langle q, P\rangle y]) \Rightarrow (P, H[x\langle q @ \checkmark, \mathbf{0}\rangle y]) \qquad [\text{S}\mathbf{1}]$$

The process P will eventually read the termination mark from the queue (triggering the deallocation of the session record from the heap):

$$(\mathsf{wait}\ y; P, H[x\langle \checkmark, \mathbf{0}\rangle y]) \Rightarrow (P, H) \qquad [\text{S}\bot]$$

Note the requirement that ✓ be the final element of the queue.

Negative Action on Write Endpoint. As hinted above for the case of executing a cut, the SAM has a kind of *write bias* insofar as the process chosen to execute in a cut is that which holds the write endpoint for the newly created session. Since CLL processes use channels bidirectionally, the role of writer and reader on a channel (and thus the holder of the write and read endpoints of the queue) may be exchanged during execution. For instance, a process P may wish to send a value v to Q and then receive a response on the same channel. However, when considering a queue-based semantics, the execution of the input action *must not* obtain the value v, intended for Q. Care is therefore needed to ensure that v is received by the holder of the read endpoint of the queue *before* P is allowed to execute its input action (and so taking over the read endpoint). This notion is captured by the following rule, where $\mathcal{A}^{-}$ denotes any process performing a negative polarity action (i.e., a wait, recv, case or, as we discuss later, a fwd x y when x is a write endpoint with a negative polarity type):

$$(\mathcal{A}^-(x), H[x\langle q, Q\rangle y]) \Mapsto (Q, H[x\langle q, \mathcal{A}^-(x)\rangle y]) \qquad [\mathrm{S}-]$$

If the executing process is to perform a negative polarity action on a write endpoint x, the SAM context switches to Q, the holder of the read endpoint y of the session, and suspends the previously running process. This will now allow for Q to perform the appropriate inputs before execution of the action $\mathcal{A}^-$ resumes.
Channel Input. The rules for recv actions are as follows:

$$\begin{array}{ll}(\mathsf{recv}\ y(w{:}+); Q, H[x\langle \mathsf{clos}(z,R)@q, P\rangle y]) \Mapsto (Q, H[w\langle \mathsf{nil}, R\rangle z][x\langle q\rangle^s y]) & [\mathrm{S}⅋_+] \\ (\mathsf{recv}\ y(w{:}-); Q, H[x\langle \mathsf{clos}(z,R)@q, P\rangle y]) \Mapsto (R, H[z\langle \mathsf{nil}, Q\rangle w][x\langle q\rangle^s y]) & [\mathrm{S}⅋_-]\end{array}$$

where $x\langle q\rangle^s y \triangleq$ if $(q = \mathsf{nil})$ then $y\langle q, P\rangle x$ else $x\langle q, P\rangle y$. The execution of an input action requires the corresponding queue to contain a process closure, denoting the process that interacts along the received channel w. In order to ensure that no inputs attempt to read from an empty queue, we must *branch* on the polarity of the communicated session (written $w{:}+$ and $w{:}-$ in the rules above): if the session has a *positive* type, then Q must take the *write* endpoint w of the newly generated queue (since Q uses the session with a dual type) and thus we execute Q and allocate a session record in the heap for the new session, with read endpoint z; if the exchanged session has a *negative* type, the converse holds and Q must take the *read* endpoint of the newly generated queue. In this scenario, we must execute R so that it may exercise its write capability on the queue and suspend Q in the new session record.

In either case, the session record for the original session is updated by removing the received message from the queue. Crucially, since processes are well-typed, if the resulting queue is empty then it must be the case that Q has no more reads to perform on the session, and so we *swap* the read and write endpoints of the session. This swap serves two purposes: first, it enables Q to perform writes if needed; secondly, and more subtly, it allows for the process, say, P, that holds the other endpoint of the queue to be resumed to perform its actions accordingly. To see how this is the case, consider that such a process will be suspended (due to rule [S−]) attempting to perform a negative action on the write endpoint of the queue. After the swap, the endpoint of the suspended process now matches its intended action. Since Q now holds the write endpoint, it will perform some number of positive actions on the session which end either in a close, which context switches to P, or until it attempts to perform a negative action on the write endpoint, triggering rule [S−] and so context switching to P.
Choice and Selection. The treatment of the additive constructs in the SAM is straightforward:

$$\begin{array}{ll}(\#\mathsf{l}\ x; Q, H[x\langle q, P\rangle y]) \Mapsto (Q, H[x\langle q@\#\mathsf{l}, P\rangle y]) & [\mathrm{S}\oplus] \\ (\mathsf{case}\ y\ \{|\#\ell \in L{:}Q_\ell\}, H[x\langle \#\mathsf{l}@q, P\rangle y]) \Mapsto (Q_{\#\mathsf{l}}, H[x\langle q\rangle^s y]) & [\mathrm{S}\&]\end{array}$$

Sending a label #l simply adds the #l to the corresponding queue and proceeds with the execution, whereas executing a case reads a label from the queue and continues execution of the appropriate branch. Since removing the label may empty the queue, we perform the same adjustment as in rules [S$\parr_+$] and [S$\parr_-$].

Forwarding. Finally, let us consider the execution of a forwarder (we overload the @ notation to also denote concatenation of queues):

$$(\mathsf{fwd}\ x^-\ y^+, H[z\langle q_1, Q\rangle x][y\langle q_2, P\rangle w]) \Rrightarrow (P, H[z\langle q_2 @ q_1, Q\rangle w]) \qquad \text{[Sfwd]}$$

A forwarder denotes the merging of two sessions x and y. Since the forwarder holds the read and write endpoints x and y, respectively, Q has written (through z) the contents of q_1, whereas the previous steps of the currently running process have written q_2. Thus, P is waiting to read $q_2 @ q_1$, justifying the rule above.

The reader may then wonder about other possible configurations of the SAM heap and how they interact with the forwarder. Specifically, what happens if y is of a positive type but a read endpoint of a queue, or, dually, if x is of a negative type but a write endpoint. The former case is *ruled out* by the SAM since the heap satisfies the invariant that any session record of the form $x{:}A\langle q, P\rangle y{:}A \in H$ is such that A must be of negative polarity or P is the inert process (which cannot be forwarded). The latter case is possible and is handled by rule [S$-$], since such a forward $\mathsf{fwd}\ x^-\ y^+$ stands for a process that wants to perform a *negative polarity* action on a *write* endpoint (or a positive action on a read endpoint).

3.1 On the Write-Bias of the SAM

Consider the following CLL process:

$$P \triangleq \mathsf{cut}\ \{P_1\ |a : \mathbf{1} \otimes \mathbf{1}|\ \{a/b\}Q_1\}$$

$$\begin{array}{ll} P_1 \triangleq \mathsf{send}\ a(y.P_2); P_3 & \quad Q_1 \triangleq \mathsf{recv}\ b(x); Q_2 \\ P_2 \triangleq \mathsf{close}\ y & \quad Q_2 \triangleq \mathsf{wait}\ x; Q_3 \\ P_3 \triangleq \mathsf{close}\ a & \quad Q_3 \triangleq \mathsf{wait}\ b; \mathsf{0} \end{array}$$

Let us walk through the execution trace of P:

(1) $(P, \emptyset) \Rrightarrow$ by [SCut]
(2) $(P_1, a\langle \mathsf{nil}, Q_1\rangle b) \Rrightarrow$ by [S$\otimes$]
(3) $(P_3, a\langle \mathsf{clos}(y, P_2), Q_1\rangle b) \Rrightarrow$ by [S$\mathbf{1}$]
(4) $(Q_1, a\langle \mathsf{clos}(y, P_2) @ \checkmark, \mathsf{0}\rangle b) \Rrightarrow$ by [S$\parr_-$]
(5) $(P_2, y\langle \mathsf{nil}, Q_2\rangle x, a\langle \checkmark, \mathsf{0}\rangle b) \Rrightarrow$ by [S$\mathbf{1}$]
(6) $(Q_2, y\langle \checkmark, \mathsf{0}\rangle x, a\langle \checkmark, \mathsf{0}\rangle b) \Rrightarrow$ by [S$\bot$]
(7) $(Q_3, a\langle \checkmark, \mathsf{0}\rangle b) \Rrightarrow$ by [S$\bot$]
(8) $(\mathsf{0}, \emptyset)$

The SAM begins in the state on line (1) above, executing the cut. Since the type of a is positive, we execute P_1, and allocate the session record, suspending Q_1, resulting in the state on line (2). Since P_1 is a write action on a write endpoint, we proceed via the [S$\otimes$] rule, resulting in the SAM configuration in line (3), executing P_3 and adding a closure containing P_2 to the session queue with write endpoint a. Executing P_3 (3), a close action, requires adding the $\checkmark$

to the queue and context switching to the process Q_1, now ready to receive the sent value. The applicable rule is now (4) [S$\parr_-$], and so execution will context switch to P_2 after creating the session record for the new session with endpoints y and x. P_2 will execute and the machine ends up in state (6) followed by (7), which consume the appropriate $\checkmark$ and deallocate the session records.

Note how after executing the send action of P_1 we eagerly execute the positive action in P_3 rather than context switching to Q_1. While in this particular process it would have been safe to execute the negative action in Q_1, switch to P_2 and then back to Q_2, we would now need to somehow context switch to P_3 *before* continuing with the execution of Q_3, or execution would be stuck. However, the relationship between P_3 and Q_2 is unclear at best. Moreover, if the continuation of Q_1 were of the form $\mathsf{wait}\ b; \mathsf{wait}\ x; \mathsf{0}$, the context switch after the execution of P_2 would have to execute P_3, or the machine would also be in a stuck state.

3.2 Illustrating Forwarding

To better illustrate the way in which $\mathsf{fwd}\ x^-\ y^+$ effectively stands for a negative action, consider the following CLL process (to simplify the execution trace we assume the existence of output and input of integers typed as $\mathsf{int} \otimes A$ and $\overline{\mathsf{int}} \parr A$, respectively, eliding the need for process closures in this example):

$$P \triangleq \mathsf{cut}\ \{P_1\ |b : \overline{\mathsf{int}} \parr \overline{\mathsf{int}} \parr \mathbf{1}|\ \{b/c\}\mathsf{cut}\ \{Q_1\ |a : \mathsf{int} \otimes \overline{\mathsf{int}} \parr \mathbf{1}|\ \{a/d\}R_1\}\}$$

$$\begin{array}{lll} P_1 \triangleq \mathsf{recv}\ b(x); P_2 & Q_1 \triangleq \mathsf{send}\ a(1); Q_2 & R_1 \triangleq \mathsf{recv}\ d(y); R_2 \\ P_2 \triangleq \mathsf{recv}\ b(z); P_3 & Q_2 \triangleq \mathsf{send}\ c(3); Q_3 & R_2 \triangleq \mathsf{send}\ d(2); R_3 \\ P_3 \triangleq \mathsf{close}\ b & Q_3 \triangleq \mathsf{fwd}\ a\ c & R_3 \triangleq \mathsf{wait}\ d; \mathsf{0} \end{array}$$

If we consider the execution of P we observe:

(1) $(P, \emptyset) \mapsto$	by [SCut]
(2) $(\mathsf{cut}\ \{Q_1\ |a|\ \{a/d\}R_1\}, c\langle \mathsf{nil}, P_1\rangle b) \mapsto$	by [SCut]
(3) $(Q_1, a\langle \mathsf{nil}, R_1\rangle d, c\langle \mathsf{nil}, P_1\rangle b) \mapsto$	by [S$\otimes$]
(4) $(Q_2, a\langle 1, R_1\rangle d, c\langle \mathsf{nil}, P_1\rangle b) \mapsto$	by [S$\otimes$]
(5) $(\mathsf{fwd}\ a\ c, a\langle 1, R_1\rangle d, c\langle 3, P_1\rangle b) \mapsto$	by [S$-$]
(6) $(R_1, a\langle 1, Q_3\rangle d, c\langle 3, P_1\rangle b) \mapsto$	by [S$\parr$]
(7) $(R_2, d\langle \mathsf{nil}, Q_3\rangle a, c\langle 3, P_1\rangle b) \mapsto$	by [S$\otimes$]
(8) $(R_3, d\langle 2, Q_3\rangle a, c\langle 3, P_1\rangle b) \mapsto$	by [S$-$]
(9) $(\mathsf{fwd}\ a\ c, d\langle 2, R_3\rangle a, c\langle 3, P_1\rangle b) \mapsto$	by [Sfwd]
(10) $(P_1, d\langle 3@2, R_3\rangle b) \mapsto$	by [S$\parr$]
(11) $(P_2, b\langle 2, R_3\rangle d) \mapsto$	by [S$\parr$]
(12) $(P_3, b\langle \mathsf{nil}, R_3\rangle d) \mapsto$	by [S$\mathbf{1}$]
(13) $(R_3, b\langle \checkmark, R_3\rangle d) \mapsto$	by [S$\bot$]
(14) $(\mathsf{0}, \emptyset)$	

The first four steps of the execution of P allocate the two session records and the writes by Q_1 and Q_2 takes place. We are now in configuration (5), where $Q_3 = \mathsf{fwd}\ a^-\ c^+$ is to execute and a is a write endpoint of a queue assigned a negative type ($\overline{\mathsf{int}} \parr \mathbf{1}$). This forwarder stands for a process performing a negative action on a write endpoint (i.e., P_1) and so context switching is required, rule [S$-$] applies and the SAM context switches to R_1, suspending Q_3 until the forward can be performed. After R_1 receives (6) and the queue endpoints a and d are swapped (7), R_2 executes and then rule [S−] applies (8), context switching back to Q_3. Since the queue endpoints are now flipped, rule [Sfwd] now applies (9), collapsing the two session records (via queue concatenation) and proceeding with the execution of P_1, P_2, P_3 and R_3 (10-14). Note the correct ordering in which the sent values are dequeued, where 3 is read before 2, as intended.

Discussion. The core execution rules for the SAM are summarized in Figure 5. At this point, the reader may wonder just how reasonable the SAM's evaluation strategy is. Our evaluation strategy is devised to be a deterministic, sequential strategy, where exactly one process is executing at any given point in time, supported by a queue-based buffer structure for channels and a heap for session records. Moreover, taking inspiration from focusing and polarized logic, we adopt a write-biased stance and prioritize (bundles of) write actions over reads, where suspended processes hold the read endpoint of queues while waiting for the writer process to fill the queue, and hold write endpoints of queues *after* filling them, waiting for the reader process to empty the queue.

While this latter point seems like a reasonable way to ensure that inputs never get stuck, it is not immediately obvious that the strategy is sound wrt the more standard (asynchronous) semantics of CLL and related languages, given that processes are free to act on multiple sessions. Thus, the write-bias of the cut rule (and the overall SAM) does not necessarily mean that the process that is chosen to execute will immediately perform a write action on the freshly cut session x. In general, such a process may perform multiple write or read actions on many other sessions before performing the write on x, meaning that multiple context switches may occur. Given this, it is not obvious that this strategy is adequate insofar as preserving the correctness properties of CLL in terms of soundness, progress and type preservation. The remainder of this paper is devoted to establishing this correspondence in a precise technical sense.

4 CLLB: A Buffered Formulation of CLL

There is a substantial gap between the language CLL, presented in an abstract algebraic style, and its operational semantics, defined by equational and rewriting systems, and an abstract machine as the SAM, a deterministic state machine manipulating several low level structures. Therefore, even if the core SAM structure and transition rules are fairly simple, proving its correctness is more challenging and technically involved, and require progressive build up. Therefore, we first bridge between CLL and SAM via a intermediate logical language CLLB, which extends CLL with a buffered cut construct.

$$\mathsf{cut}\ \{P\ |a : A\ [q]\ b : B|\ Q\}$$

The buffered cut construct models interaction via a "message queue" with two polarised endpoints a and b, held respectively by the processes P and Q. A polarised endpoint has the form x or $\overline{x}$. The endpoint marked $\overline{x}$ is the only allowing writes, the unmarked y is the only one allowing reads, exactly one of

$$
\begin{array}{ll}
(\mathsf{cut}\ \{P\ |x : A^+|\ Q\}, H) \Rrightarrow (P, H[x\langle \mathsf{nil}, \{y/x\}Q\rangle y]) & [\text{SCut}] \\
(\mathsf{fwd}\ x\ y, H[z\langle q_1, Q\rangle x][y\langle q_2, P\rangle w]) \Rrightarrow (P, H[z\langle q_2@q_1, Q\rangle w]) & [\text{Sfwd}] \\
(\mathsf{close}\ x, H[x\langle q, P\rangle y]) \Rrightarrow (P, H[x\langle q@\checkmark, \mathsf{0}\rangle y]) & [\text{S}\mathbf{1}] \\
(\mathsf{wait}\ y; P, H[x\langle \checkmark, \mathsf{0}\rangle y]) \Rrightarrow (P, H) & [\text{S}\bot] \\
(\mathcal{A}^-(x), H[x\langle q, Q\rangle y]) \Rrightarrow (Q, H[x\langle q, \mathcal{A}^-(x)\rangle y]) & [\text{S}-] \\
(\mathsf{send}\ x(z.R); Q, H[x\langle q, P\rangle y]) \Rrightarrow (Q, H[x\langle q@\mathsf{clos}(z, R), P\rangle y]) & [\text{S}\otimes] \\
(\mathsf{recv}\ y(w : +); Q, H[x\langle \mathsf{clos}(z, R)@q, P\rangle y]) \Rrightarrow (Q, H[w\langle \mathsf{nil}, R\rangle z][x\langle q\rangle^s y]) & [\text{S}⅋] \\
(\mathsf{recv}\ y(w : -); Q, H[x\langle \mathsf{clos}(z, R)@q, P\rangle y]) \Rrightarrow (R, H[z\langle \mathsf{nil}, Q\rangle w][x\langle q\rangle^s y]) & [\text{S}⅋] \\
(\#\mathsf{l}\ x; Q, H[x\langle q, P\rangle y]) \Rrightarrow (Q, H[x\langle q@\#\mathsf{l}, P\rangle y]) & [\text{S}\oplus] \\
(\mathsf{case}\ y\ \{|\#\ell \in L{:}Q_\ell\}, H[x\langle \#\mathsf{l}@q, P\rangle y]) \Rrightarrow (Q_{\#\mathsf{l}}, H[x\langle q\rangle^s y]) & [\text{S}\&] \\
N.B. : x\langle q\rangle^s y \triangleq \text{if } (q = \mathsf{nil}) \text{ then } y\langle q, P\rangle x \text{ else } x\langle q, P\rangle y &
\end{array}
$$

Fig. 5: The core SAM Transition Rules

the two endpoints is marked. The endpoints types A, B are of course related but do not need to be exact duals, the type of the writer endpoint may be advanced in time wrt the type of the reader endpoint, reflecting the messages already enqueued but not yet consumed. If the queue is empty, we have $A = \overline{B}$. Thus a buffered cut with empty queue corresponds to the basic cut of CLL.

$$\mathsf{cut}\ \{P\ |x : A|\ Q\} \equiv \mathsf{cut}\ \{P\ |\overline{x} : A\ [\mathsf{nil}]\ y : \overline{A}|\ \{y/x\}Q\} \qquad (A+)$$

The queue q stores values V defined by

$$
\begin{array}{lllll}
V ::= & \checkmark & (\text{Close token}) & |\ \ \#\mathsf{l} & (\text{Selection Label}) \\
| & \mathsf{clos}(x, P) & (\text{Linear Closure}) & |\ \ \mathsf{clos!}(x, P) & (\text{Exponential Closure})
\end{array}
$$

$$q ::= \mathsf{nil}\ |\ V\ |\ V@q \quad (\text{Queue})$$

We use @ to also denote (associative) concatenation operation of queues, with unit nil. Enqueue and dequeue operations occur respectively on the lhs and rhs.

The type system CLLB is obtained from CLL by replacing [TCut] with the typing rules (and symmetric ones) in Fig. 6. We distinguish the type judgements as $P \vdash \Delta; \Gamma$ for CLL and $P \vdash^{\mathsf{B}} \Delta; \Gamma$ for CLLB. The [TCutB] rule sets the endpoints mode based in the cut type polarity, applicable whenever the queue is empty. The remaining rules relate queue contents with their corresponding (positive action) processes. For instance, rule [Tcut-$\otimes$] can be read bottom-up as stating that typing processes mediated by a queue containing a process closure $\mathsf{clos}(y, R)$ amounts to typing the process that will emit the session y (bound to R), interacting with the queue with the closure removed. Rules [Tcut-$\oplus$] and [Tcut!] apply a similar principle to the other possible queue contents. In [Tcut-$\mathbf{1}$] and [Tcut!] the write endpoint is typed $\emptyset$, as the sender has terminated ($\mathsf{0}$).

$$\frac{P \vdash^{\mathsf{B}} \Delta', x : \overline{A}; \Gamma \quad Q \vdash^{\mathsf{B}} \Delta, y : A; \Gamma}{\mathsf{cut}\ \{P\ |x : \overline{A}\ [\mathsf{nil}]\ \overline{y} : A|\ Q\}\ \vdash^{\mathsf{B}} \Delta', \Delta; \Gamma}\ (\text{A positive}) \quad [\text{TcutB}]$$

$$\frac{\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x} : \mathbf{1}\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{0\ |\overline{x} : \emptyset[q@\checkmark]y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\ [\text{Tcut-}\mathbf{1}]$$

$$\frac{\mathsf{cut}\ \{\mathsf{send}\ x(y.R); P\ |\overline{x} : T \otimes A\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{P\ |\overline{x} : A\ [q@\mathsf{clos}(y, R)]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\ [\text{Tcut-}\otimes]$$

$$\frac{\mathsf{cut}\ \{\#\mathsf{l}\ x; P\ |\overline{x} : \oplus_{\ell \in L} A_\ell\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{P\ |\overline{x} : A_{\#\mathsf{l}}\ [q@\#\mathsf{l}]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\ [\text{Tcut-}\oplus]$$

$$\frac{\mathsf{cut}\ \{!x(z); P\ |\overline{x} : !A\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{0\ |\overline{x} : \emptyset\ [q@\mathsf{clos!}(z, P)]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\ [\text{Tcut!}]$$

Fig. 6: Additional typing rules for CLLB.

$$\mathsf{cut}\ \{Q\ |a : A[q]b : B|\ P\}\ \equiv^{\mathsf{B}} \mathsf{cut}\ \{Q\ |b : B[q]a : A|\ P\} \quad [\text{comm}]$$

$$\mathsf{cut}\ \{P\ |x[q]y|\ (Q\ ||\ R)\} \equiv^{\mathsf{B}} (\mathsf{cut}\ \{P\ |x[q]y|\ Q\})\ ||\ R \quad [\text{CM}]$$

$$P\ |x[q]z|\ (\mathsf{cut}\ \{Q\ |y[p]w|\ R\}) \equiv^{\mathsf{B}} \mathsf{cut}\ \{(\mathsf{cut}\ \{P\ |x[q]z|\ Q\})\ |y[p]w|\ R\} \quad [\text{CC}]$$

$$\mathsf{cut}\ \{P\ |z[q]w|\ (\mathsf{cut!}\ \{y.Q\ |!x|\ R\})\} \equiv^{\mathsf{B}} \mathsf{cut!}\ \{y.Q\ |!x|\ (\mathsf{cut}\ \{P\ |z[q]w|\ R\})\} \quad [\text{CC!}]$$

$$\mathsf{cut!}\ \{y.P\ |!x|\ (\mathsf{cut}\ \{Q\ |z[q]w|\ R\})\} \equiv^{\mathsf{B}} \mathsf{cut}\ \{(y.P\ |!x|\ Q)\ |z[q]w|\ (\mathsf{cut!}\ \{y.P\ |!x|\ R\}\}) \quad [\text{D-C!}]$$

Fig. 7: Additional structural congruence rules for CLLB.

Structural congruence for B (noted $\equiv^{\mathsf{B}}$) is obtained by extending $\equiv$ with commutative conversions for the buffered cut, listed in Fig. 7. The following provisos apply: [CM] $y \in \mathsf{fn}(Q)$; in [CC] $y, z \in \mathsf{fn}(Q)$; in [CC!] $x \notin \mathsf{fn}(P)$. Accordingly, reduction for B (noted $\rightarrow^{\mathsf{B}}$) is obtained by replacing the $\rightarrow$ rules [fwd], [$\mathbf{1}\bot$], [$\otimes⅋$] and [$\oplus\&$] by the rules in Fig. 8. Essentially each principal cut reduction rule of CLL is replaced by a pair of "positive" ($\rightarrow_p$) / "negative" ($\rightarrow_n$) reduction rules that allow processes to interact asynchronously via the queue, that is, positive process actions (corresponding to positive types) are non-blocking. For example, the rule [$\otimes$] for send appends a session closure to the tail of the queue (rhs) and the rule for receive pops a session closure from the head of the queue (lhs). Notice that positive rules are enabled only if the relevant endpoint is in write mode ($\overline{x}$), and negative rules are enabled only if the relevant endpoint is in read mode (y). In [⅋] above the target cuts endpoint polarities depends on the types of the composed processes. To uniformly express the appropriate marking of endpoint polarities we define some convenient abbreviations:

$$\mathsf{cut}\ \{Q\ |\overline{z}\ [q_1]\ x|\ \mathsf{fwd}\ x\ y\ |\overline{y}\ [q_2]\ w|\ P\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{z}\ [q_2@q_1]\ w|\ P\} \qquad [\text{fwdp}]$$

$$\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x}\ [q]\ y|\ Q\} \to^{\mathsf{B}} \mathsf{cut}\ \{0\ |\overline{x}\ [q@\checkmark]\ y|\ Q\} \qquad [\mathbf{1}]$$

$$\mathsf{cut}\ \{0\ |\overline{x}\ [\checkmark]\ y|\ \mathsf{wait}\ y; P\} \to^{\mathsf{B}} P \qquad [\bot]$$

$$\mathsf{cut}\ \{\mathsf{send}\ x(z.P); Q\ |\overline{x}\ [q]\ y|\ R\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q@\mathsf{clos}(z,P)]|\ y|\ R\} \qquad [\otimes]$$

$$\mathsf{cut}\ \{Q\ |\overline{x}\ [\mathsf{clos}(z,P)@q]\ y|\ \mathsf{recv}\ y(w); R\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q]\ y|\ \mathsf{cut}\ \{P\ |z\ [\mathsf{nil}]\ w|\ R\}^{\mathsf{p}}\}^{\mathsf{p}} \qquad [⅋]$$

$$\mathsf{cut}\ \{\#\mathsf{l}\ x; P\ |\overline{x}\ [q]\ y|\ R\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q@\#\mathsf{l}]|\ y|\ R\} \qquad [\oplus]$$

$$\mathsf{cut}\ \{Q\ |\overline{x}\ [\mathsf{l}@q]\ y|\ \mathsf{case}\ y\ \{|\#\ell \in L{:}P_\ell\}\ \} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |x\ [q]\ y|\ P_{\#\mathsf{l}}\}^{\mathsf{p}} \qquad [\&]$$

$$\mathsf{cut}\ \{!x(z); P\ |\overline{x}\ [q]\ y|\ Q\} \to^{\mathsf{B}} \mathsf{cut}\ \{0\ |\overline{x}\ [q@\mathsf{clos!}(z,P)]|\ y|\ Q\} \qquad [!]$$

$$\mathsf{cut}\ \{0\ |\overline{x}\ [\mathsf{clos!}(y,P)]\ y|\ ?y; Q\} \to^{\mathsf{B}} \mathsf{cut!}\ \{y.P\ |!x|\ Q\} \qquad [?]$$

Fig. 8: Reduction $P \to^{\mathsf{B}} Q$.

Definition 4.1 (Setting polarities).

$$\begin{aligned}
\mathsf{cut}\ \{Q\ |a : A[\mathsf{nil}]b : B|\ P\}^{\mathsf{p}} &\triangleq \textit{if } +A \textit{ then } \mathsf{cut}\ \{Q\ |\overline{a} : A[\mathsf{nil}]b : B|\ P\} \\
&\qquad \textit{else } \mathsf{cut}\ \{Q\ |a : A[\mathsf{nil}]\overline{b} : B|\ P\} \\
\mathsf{cut}\ \{Q\ |\overline{a} : A[q]b : B|\ P\}^{\mathsf{p}} &\triangleq \mathsf{cut}\ \{Q\ |\overline{a} : A[q]b : B|\ P\} \quad (q \neq \mathsf{nil})
\end{aligned}$$

The following definition then formalizes the intuition given above about how to encode processes of CLL into processes of CLLB.

Definition 4.2 (Embedding). *Let $P \vdash \Delta; \Gamma$. $P^\dagger$ is the* B *process such that*

$$(\mathsf{cut}\ \{P\ |x : A|\ Q\})^\dagger \triangleq \mathsf{cut}\ \{P^\dagger\ |x : A\ [\mathsf{nil}]\ y : \overline{A}|\ (\{y/x\}Q)^\dagger\}^{\mathsf{p}}$$

homomorphically defined in the remaining constructs. Clearly $P^\dagger \vdash^{\mathsf{B}} \Delta; \Gamma$.

4.1 Preservation and Progress for CLLB

In this section, we prove basic safety properties of CLLB: Preservation (Theorem 4.1) and Progress (Theorem 4.2). To reason about type derivations involving buffered cuts, we formulate some auxiliary inversion principles that allow us, by aggregating sequences of application of [TCut-$*$] rules of CLLB, to talk in a uniform way about typing of values in queues and typing of processes connected by queues. To assert typing of queue values c we use judgments the form $\Gamma; \Delta \vdash c{:}E$, where E is a either a type or a one hole type context, defined by

$$E ::= \square \mid T \mid T ⅋ E \mid \ \&_{\ell \in L}\, E_\ell$$

where in $\&_{\ell \in L} E_\ell$ only branch type $E_{\#\mathsf{l}}$ for some selected label $\#\mathsf{l} \in L$ is a one hole context (to plug the continuation type); only the branch chosen by the selected label in a queue is relevant to type next queue values. We identify the selected branch in the type by tagging it with the corresponding label $\#\mathsf{l}$ thus $\&_{\ell \in L} E_\ell[\#\mathsf{l}]$. We then introduce the following typing rules for queue values.

Definition 4.3 (Typing of Queue Values).

$$\frac{}{\Gamma; \vdash_{\mathtt{val}} \checkmark : \bot} \qquad \frac{P \vdash^{\mathsf{B}} \Delta, z : T; \Gamma}{\Gamma; \Delta \vdash_{\mathtt{val}} \mathsf{clos}(z, P) : \overline{T} \parr E}$$

$$\frac{\&_{\ell \in L} E_\ell[\#\mathsf{l}]}{\Gamma; \Delta \vdash_{\mathtt{val}} \#\mathsf{l} : \&_{\ell \in L} E_\ell} \qquad \frac{P \vdash^{\mathsf{B}} z : A; \Gamma}{\Gamma; \vdash_{\mathtt{val}} \mathsf{clos!}(z, P) : ?\overline{A}}$$

Given a sequence of k one hole queue value types E_i and a type A, we denote by $\mathbf{E}_k; A$ the type $E_1[E_2[...E_k[A]]]$. Queue value types allow us to talk in a uniform way about the type a receiver processes compatible with the types of enqueued values, as characterized by the following Lemma 4.1 and Lemma 4.2.

Lemma 4.1 (Non-full). *For $P \neq 0$ the rule below is admissible and invertible:*

$$\frac{P \vdash^{\mathsf{B}} \Delta_P, x{:}A; \Gamma \quad Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma \quad q = \overline{c_k} \quad B = \mathbf{E}_k; \overline{A} \quad \Gamma; \Delta_i \vdash c_i{:}E_i \quad -B}{\mathsf{cut}\ \{P \mid \overline{x}{:}A\ [q]\ y{:}B \mid Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_1, ..., \Delta_k; \Gamma}$$

Notice that a session type, as defined by a CLL proposition, may terminate in either $\mathbf{1}$, $\bot$ or an exponential type $!A/?A$. We then also have

Lemma 4.2 (Full). *The proof rules below are admissible:*

$$\frac{Q \vdash^{\mathsf{B}} \Delta_Q, y : B; \Gamma \quad \Gamma; \Delta_i \vdash c_i : E_i \quad B = \mathbf{E}_k; \bot \quad c_k = \checkmark \quad -B}{\mathsf{cut}\ \{0 \mid \overline{x} : \emptyset\ [\overline{c_k}]\ y : B \mid Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_1, \ldots, \Delta_k; \Gamma}$$

$$\frac{Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma \quad \Gamma; \Delta_i \vdash c_i{:}E_i \quad B = \mathbf{E}_{k-1}; C \quad \Gamma \vdash c_k = \mathsf{clos!}(z, R){:}C \quad -B}{\mathsf{cut}\ \{0 \mid \overline{x} : \emptyset\ [\overline{c_k}]\ y : B \mid Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_1, \ldots, \Delta_k, \Gamma}$$

Moreover, one of them must apply for inverting the judgment in the conclusion.

Theorem 4.1 (Preservation). *Let $P \vdash^{\mathsf{B}} \Delta; \Gamma$.*
(1) If $P \equiv^{\mathsf{B}} Q$, then $Q \vdash^{\mathsf{B}} \Delta; \Gamma$. (2) If $P \to^{\mathsf{B}} Q$, then $Q \vdash^{\mathsf{B}} \Delta; \Gamma$.

A process P is *live* if and only if $P = \mathcal{C}[Q]$, for some static context $\mathcal{C}$ (the hole lies within the scope of static constructs mix, cut) and Q is an action process. We first show that a live process either reduces or offers an interaction on a free name. The observability predicate defined in Fig. 9 (cf. [63]) characterises interactions of a process with the environment.

Lemma 4.3 (Liveness). *Let $P \vdash^{\mathsf{B}} \Delta; \Gamma$ be live. Either $P \downarrow_x$ or $P \to^{\mathsf{B}}$.*

Theorem 4.2 (Progress). *Let $P \vdash \emptyset; \emptyset$ be a live process. Then, $P \to^{\mathsf{B}}$.*

$$\frac{}{\mathsf{fwd}\ x\ y \downarrow_x}\,[\text{fwd}] \quad \frac{s(\mathcal{A}) = x}{\mathcal{A} \downarrow_x}\,[\mathcal{A}] \quad \frac{P \equiv Q \quad Q \downarrow_x}{P \downarrow_x}\,[\equiv] \quad \frac{P \downarrow_x}{(P \mid\mid Q) \downarrow_x}\,[\text{mix}]$$

$$\frac{P \downarrow_x \quad x \neq y}{(P\ |y[q]x|\ Q) \downarrow_x}\,[\text{cut}] \quad \frac{Q \downarrow_x \quad x \neq y}{(z.P\ |!y|\ Q) \downarrow_x}\,[\text{cut!}]$$

Fig. 9: Observability Predicate $P \downarrow_x$.

4.2 Correspondence between CLL and CLLB

In this section we establish the correspondence between reduction in CLL and CLLB, proving that the two languages simulate each other in a tight sense. Intuitively, the correspondence shows that CLLB allows some positive actions to be buffered ahead of reception, while in CLL a single positive action synchronises with the corresponding dual in one step, or a forward reduction takes place.

We write a reduction $P \to^{\mathsf{B}} Q$ as $P \to^{\mathsf{B}p} Q$ if the reduced action is positive, $P \to^{\mathsf{B}n} Q$ if the reduced action is negative (we consider [call] negative), $P \to^{\mathsf{B}a} Q$ if the reduced action is a forwarder, and $P \to^{\mathsf{Bap}} Q$ if the reduced action is positive or a forwarder. We also write $P \to^{\mathsf{Br}} Q$ for positive action followed by a matching negative action on the same cut with an initially empty queue.

Lemma 4.4. *The following commutations of reductions hold.*

1. *Let $P_1 \to^{\mathsf{B}p} S \to^{\mathsf{B}n} P_2$. Either $P_1 \to^{\mathsf{Br}} P_2$, or $P_1 \to^{\mathsf{B}n} S' \to^{\mathsf{B}p} P_2$ for some S'.*
2. *Let $P_1 \to^{\mathsf{B}a} S \to^{\mathsf{B}n} P_2$. Then $P_1 \to^{\mathsf{B}n} S' \to^{\mathsf{B}a} P_2$ for some S'.*
3. *If $P_1 \to^{\mathsf{Bap}} S \to^{\mathsf{B}n} P_2$, either $P_1 \to^{\mathsf{Br}} P_2$, or $P_1 \to^{\mathsf{B}n} S' \to^{\mathsf{Bap}} P_2$ for some S'.*
4. *Let $P_1 \to^{\mathsf{Bap}} N \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} S \to^{\mathsf{Br}} P_2$. Either $P_1 \overset{\epsilon}{\to}^{\mathsf{B}a} N$ or $P_1 \to^{\mathsf{Br}} S' \to^{\mathsf{Bap}} P_2$ for some S'.*

Lemma 4.5 (Simulation). *Let $P \vdash \emptyset; \emptyset$. If $P \to Q$ then $P^\dagger \Rightarrow^{\mathsf{B}} Q^\dagger$.*

Proof. Each cut reduction of CLL is either simulated by two reduction steps of B in sequence or by a [fwd] reduction.

The following lemma identifies that in CLLB, a sequence of positive actions (or forwards) followed by a negative action can always be commuted either by pulling out the negative action first, followed by the sequence of positive actions and forwards; having the negative action follow a positive action on the same channel and then performing the remaining actions; or by first performing a sequence of forward actions, the output and input on the relevant session and then the remaining actions.

Lemma 4.6 (Simulation). *Let $P \vdash^{\mathsf{B}} \emptyset; \emptyset$. If $P \Rightarrow^{\mathsf{Bap}} \to^{\mathsf{B}n} Q$ then (1) $P \to^{\mathsf{B}n} R$ and $R \Rightarrow^{\mathsf{Bap}} Q$ for some R, or; (2) $P \to^{\mathsf{Br}} R$ and $R \Rightarrow^{\mathsf{Bap}} Q$ for some R, or; (3) $P \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} \to^{\mathsf{Br}} R$ and $R \Rightarrow^{\mathsf{Bap}} Q$ for some R.*

Theorem 4.3 (Operational correspondence CLL-CLLB). *Let $P \vdash \emptyset; \emptyset$.*

1. *If $P \Rightarrow R$ then $P^\dagger \Rightarrow^{\mathsf{B}} R^\dagger$.*

2. If $P^\dagger (\Rightarrow^{\mathsf{Bap}} \rightarrow^{\mathsf{B}n})^* Q$ *then there is* R *such that* $P \Rightarrow R$ *and* $R^\dagger \Rightarrow^{\mathsf{Bap}} Q$*.*

Due to the progress property for CLLB (Theorem 4.2) and because queues are bounded by the size of positive/negative sections in types, after a sequence of positive or forwarder reductions a negative reduction consuming a queue value must occur. Theorem 4.3(2) states that every reduction sequence in CLLB is simulated by a reduction sequence in CLL up to some anticipated forwarding and buffering of positive actions. Our results imply that every reduction path in CLLB maps to a reduction path in CLL in which every negative reduction step in the former is mapped, in order, to a cut reduction step in the latter.

5 Correctness of the core SAM

We now prove that every execution trace of the core SAM defined in Fig. 5 represents a correct process reduction sequence CLLB (and therefore of CLL, in the light of Theorem 4.3), first for the language without exponentials and mix, which will be treated in Section 6. In what follows, we annotate endpoints of session records with their types (e.g. as $x{:}A\langle q, P\rangle y{:}B$), these annotations are not needed to guide the operation of the SAM, but convenient for the proofs; they will be omitted when not relevant or are obvious from the context. We first define a simple encoding of well-typed CLLB processes to SAM states.

Definition 5.1 (Encode). *Given* $P \vdash^{\mathsf{B}} \emptyset$ *we define* $enc(P) = \mathcal{C}$ *as* $enc(P, \emptyset) \overset{\mathsf{cut}*}{\Rrightarrow} \mathcal{C}$ *where* $enc(P, H) \overset{\mathsf{cut}*}{\Rrightarrow} \mathcal{C}$ *is defined by the rules*

$$\frac{enc(P(x), H[x{:}A\langle q, Q\rangle y{:}B]) \overset{\mathsf{cut}*}{\Rrightarrow} \mathcal{C}}{enc(\mathsf{cut}\ \{P\ |\overline{x}{:}A[q]\ y{:}B]|\ Q\}, H) \overset{\mathsf{cut}*}{\Rrightarrow} \mathcal{C}} \quad (A+)$$

$$\frac{enc(Q(y), H[x{:}A\langle q, P\rangle y{:}B]) \overset{\mathsf{cut}*}{\Rrightarrow} \mathcal{C}}{enc(\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q]\ y{:}B]|\ Q\}, H) \overset{\mathsf{cut}*}{\Rrightarrow} \mathcal{C}} \quad (A- \text{ or } P = \mathsf{0})$$

$$enc(\mathcal{A}, H) \overset{\mathsf{cut}*}{\Rrightarrow} (\mathcal{A}, H) \qquad (A \in \mathcal{A})$$

Notice that $enc(P)$ maximally applies the SAM execution rule for cut to $(P, \emptyset)$ until an action is reached. Clearly, for any $P \vdash^{\mathsf{B}} \emptyset$, if $enc(P) = \mathcal{C}$ then $P \Rrightarrow^* \mathcal{C}$. Also, if all cuts in a state C have empty queues then there is a process Q of CLL such that $enc(Q^\dagger) = C$. We then have

Theorem 5.1 (Soundness wrt CLLB). *Let* $P \vdash^{\mathsf{B}} \emptyset$*.*
If $enc(P) \Rrightarrow D \overset{\mathsf{cut}*}{\Rrightarrow} \mathcal{C}$ *then there is* Q *such that* $P \rightarrow \cup \equiv Q$ *and* $\mathcal{C} = enc(Q)$*.*

We can then combine soundness with the operational correspondence between CLL and CLLB (Theorem 4.3) to obtain an overall soundness result for the SAM with respect to CLL:

Theorem 5.2 (Soundness wrt CLL). *Let* $P \vdash^{\mathsf{B}} \emptyset$*.*
1. If $enc(P) \overset{*}{\Rrightarrow} \overset{\mathsf{cut}*}{\Rrightarrow} C$ *there is* Q *such that* $P \Rightarrow \cup \equiv Q$ *and* $\mathcal{C} = enc(Q)$*.*
2. Let $P \vdash \emptyset$*. If* $enc(P^\dagger) \overset{*}{\Rrightarrow} enc(Q^\dagger)$ *then* $P \Rightarrow Q$*.*

In Definition 5.2 we identify readiness, the fundamental invariant property of SAM states, key to prove progress of its execution strategy. Readiness means that any running process holding an endpoint of negative type, and thus attempting to execute a negative action (e.g., a receive or offer action) on it, will always find an appropriate value (resp. a closure or a label) to be read in the appropriate session queue. No busy waiting or context switching will be necessary since the sequential execution semantics of the SAM enforces that all actions corresponding to a positive section of a session type have always been enqueued by the "caller" process before the "callee" takes over. As discussed in Section 3 it might not seem obvious whether all such input endpoints, (including endpoints moved around via send / receive interactions), always refer to non-empty queues.

Readiness must also be maintained by processes suspended in session records, even if a suspended process waiting on a read endpoint will not necessarily have the corresponding queue already populated. Intuitively, a process P is (H, N)-ready if all its "reads" in the input channels (except those in N) will be matched by values already stored in the corresponding session queue.

Definition 5.2 (Ready). *Process P is H, N-ready if for all $y \in \mathsf{fn}(P) \setminus N$ and $x : A\langle q, R\rangle y \in H$ then A is negative or void. We abbreviate $H, \emptyset$-ready by H-ready. Heap H is ready if, for all $x\langle q, R\rangle y \in H$, the following conditions hold:*

1. *if $R(y)$ then R is $H, \{y\}$-ready*
2. *if $R(x)$ then R is H-ready*
3. *if $\mathsf{clos}(z : -, R) \in q$, R is $H, \{z\}$-ready.*
4. *if $\mathsf{clos}(z : +, R) \in q$, R is H-ready.*

State $C = (P, H)$ is ready if H is ready and P is H-ready.

Lemma 5.1 (Readiness). *Let $P \vdash \emptyset$ and $(P, \emptyset) \overset{*}{\Rightarrow} S$. Then $\mathcal{S}$ is ready.*

Theorem 5.3 (Progress). *Let $P \vdash^{\mathsf{B}} \emptyset$ and P live. Then $enc(P) \Rightarrow S'$.*

6 The SAM for full CLL

In this section, we complete our initial presentation of the SAM, in particular, we introduce support for the exponentials, allowing the machine to compute with non-linear values, and a selective concurrency semantics. We have delayed the introduction of an environment structure for the SAM, to make the presentation easier to follow. However, this was done at the expense of a more abstract formalisation of the operational semantics, making use of α-conversion, and overloading language syntax names as heap references for allocated session records.

The SAM actually relies on environment-based implementation of name management, presented in Fig. 6. A SAM state is then a triple $(\mathcal{E}, P, H)$ where $\mathcal{E}$ is an environment that maps each free name of the code P into either a closure or a heap record endpoint. These heap references are freshly allocated and unique, thus avoiding any clashes and enforcing proper static scoping. Closures, representing suspended linear ($\mathsf{clos}(z, \mathcal{E}, P)$) and exponential behaviour ($\mathsf{clos!}(z, \mathcal{E}, P)$), pair the code in its environment, and we expect the following structural safety conditions for name biding in configurations to hold.

$$\begin{array}{lll}
S & ::= (\mathcal{E}, P, H) & \text{State} \\
H & ::= \mathit{Ref} \to \mathit{SessionRec} & \text{Heap} \\
\mathit{SessionRec} & ::= x\langle q, \mathcal{E}, P\rangle y & \\
q & ::= \textsf{nil} \mid \mathit{Val}@q & \text{Queue} \\
\mathit{Val} & ::= \checkmark & \text{Close token} \\
 & \mid \;\; \#\textsf{l} & \text{Choice label} \\
 & \mid \;\; \textsf{clos}(x, \mathcal{E}, P) & \text{Linear Closure} \\
 & \mid \;\; \textsf{clos!}(x, \mathcal{E}, P) & \text{Exponential Closure} \\
\mathcal{E}, \mathcal{G}, \mathcal{F} & ::= \mathit{Name} \to (\mathit{Ref} \cup \mathit{Val}) & \text{Environment}
\end{array}$$

Fig. 10: The SAM

Definition 6.1 (Closure).
A process P is $(\mathcal{E}, N)$-closed if $\textsf{fn}(P) \setminus N \subseteq dom(\mathcal{E})$, and $\mathcal{E}$-closed if $(\mathcal{E}, \emptyset)$-closed. Environment $\mathcal{E}$ is H-closed if for all $x \in dom(\mathcal{E})$ if $\mathcal{E}(x)$ is a reference then $x \in H$, if $\mathcal{E}(x) = \textsf{clos!}(z, \mathcal{F}, R)$ then $\mathcal{F}$ is H-closed and R is $(\mathcal{F}, \{z\})$-closed. Heap H is closed if for all $x\langle q, \mathcal{G}, Q\rangle y \in H$, $\mathcal{G}$ is H-closed, Q is $\mathcal{G}$-closed, and for all $\textsf{clos}(z, \mathcal{F}, R) \in q$ and $\textsf{clos!}(z, \mathcal{F}, R) \in q$, $\mathcal{F}$ is H closed and R is $(\mathcal{F}, \{z\})$-closed. State $(\mathcal{E}, P, H)$ is closed if H is closed, $\mathcal{E}$ is H-closed, and P is $\mathcal{E}$-closed.

In Figure 6 we present the environment-based execution rules for the SAM. All rules except those for exponentials have already been essentially presented in Fig. 5 and discussed in previous sections. The only changes to those rules are due to the presence of environments, which at all times record the bindings for free names in the code. Overall, we have

Lemma 6.1. *Let $P \vdash^{\mathsf{B}} \emptyset; \emptyset$. For all S such that $(P, \emptyset, \emptyset) \overset{*}{\Rightarrow} S$, S is closed.*

We discuss the SAM rules for the exponentials. Values of exponential type are represented by exponential closures $\textsf{clos!}(z, \mathcal{F}, R)$. Recall that a session type may terminate in either type $\mathbf{1}$, type $\bot$ or in an exponential type $!A/?A$ (cf. 4.2). So, the (positive) execution rule [S!] is similar to rule [S$\mathbf{1}$]: it enqueues the closure representing the replicated process, and switches context, since the session terminates (cf. [!] Fig. 8). The execution rule [S?] is similar to rule [S$\otimes$]: it pops a closure from the queue (which, in this case, always becomes empty), and instead of using it immediately, adds it to the environment to become persistently available to client code (cf. reduction rule [S?] Fig. 8). Any such closure representing a replicated process may be called by client code with transition rule [Scall], which essentially creates a new linear session composed by cut with the client code, similarly to [S$\otimes$]. Rule [SCall] operates with some similarity to rule [S$\otimes$]: instead of activating a linear closure popped from the queue, it activate an exponential closure fetched from the environment.

We extend the *enc* map to the exponential cut and environment states $(\mathcal{E}, P, H)$ by adapting Definition 5.1, and adding the clause:

$(\mathcal{E}, \mathsf{cut}\ \{P\ |\overline{x} : A\ [\mathsf{nil}]\ y : B|\ Q\}, H) \mapsto (\mathcal{G}, P, H[a\langle q, \mathcal{F}, Q\rangle b])$ [SCut]
$a, b = \mathsf{new}, \mathcal{G} = \mathcal{E}\{a/x\}, \mathcal{F} = \mathcal{E}\{b/y\}$

$(\mathcal{E}, \mathsf{close}\ x, H[a\langle q, \mathcal{F}, P\rangle b]) \mapsto (\mathcal{F}, P, H[a\langle q@\checkmark, \emptyset, \mathsf{0}\rangle b])$ [S**1**]
$a = \mathcal{E}(x)$

$(\mathcal{E}, \mathsf{fwd}\ x\ y, H[c\langle q_1, \mathcal{G}, Q\rangle a][b\langle q_2, \mathcal{F}, P\rangle d]) \mapsto (\mathcal{F}, P, H[c\langle q_2@q_1, \mathcal{G}, Q\rangle d])$ [Sfwd]
$a = \mathcal{E}(x), b = \mathcal{E}(y)$

$(\mathcal{E}, \mathsf{wait}\ y; P, H[a\langle \checkmark, \emptyset, \mathsf{0}\rangle b]) \mapsto (\mathcal{E}, P, H)$ [S$\perp$]
$b = \mathcal{E}(y)$

$(\mathcal{E}, \mathcal{A}^-(x), H[a\langle q, \mathcal{G}, Q\rangle b]) \mapsto (\mathcal{G}, Q, H[a\langle q, \mathcal{E}, \mathcal{A}^-(x)\rangle b])$ [S$-$]
$a = \mathcal{E}(x)$

$(\mathcal{E}, \mathsf{send}\ x(z.R); Q, H[a\langle q, P\rangle b]) \mapsto$
$(\mathcal{E}, Q, H[a\langle q@\mathsf{clos}(z, \mathcal{E}, R), P\rangle b])$ [S$\otimes$]
$a = \mathcal{E}(x)$

$(\mathcal{E}, \mathsf{recv}\ y(w{:}+); Q, H[a\langle \mathsf{clos}(z, \mathcal{F}, R)@q, \mathcal{G}, P\rangle b]) \mapsto$
$(\mathcal{E}', Q, H[e\langle \mathsf{nil}, \mathcal{F}', R\rangle f][a\langle q\rangle^s b])$ [S$⅋+$]
$e, f = \mathsf{new}, b = \mathcal{E}(y), \mathcal{E}' = \mathcal{E}\{e/w\}, \mathcal{F}' = \mathcal{F}\{f/z\}$

$(\mathcal{E}, \mathsf{recv}\ y(w{:}-); Q, H[a\langle \mathsf{clos}(z, \mathcal{F}, R)@q, \mathcal{G}, P\rangle b]) \mapsto$
$(\mathcal{F}', R, H[e\langle \mathsf{nil}, \mathcal{E}', Q\rangle f][a\langle q\rangle^s b])$ [S$⅋-$]
$e, f = \mathsf{new}, b = \mathcal{E}(y), \mathcal{F}' = \mathcal{F}\{e/z\}, \mathcal{E}' = \mathcal{E}\{f/w\}$

$(\mathcal{E}, \#\mathsf{l}\ x; Q, H[a\langle q, \mathcal{G}, P\rangle b]) \mapsto (\mathcal{E}, Q, H[a\langle q@\#\mathsf{l}, \mathcal{G}, P\rangle b])$ [S$\oplus$]
$a = \mathcal{E}(x)$

$(\mathcal{E}, \mathsf{case}\ y\ \{|\#\ell \in L{:}Q_\ell\}, H[a\langle \#\mathsf{l}@q, \mathcal{G}, P\rangle b]) \mapsto (\mathcal{E}, Q_{\#\mathsf{l}}, H[a\langle q\rangle^s b])$ [S&]
$b = \mathcal{E}(y)$

$(\mathcal{E}, !x(z); Q, H[a\langle q, \mathcal{G}, P\rangle b]) \mapsto (\mathcal{G}, P, H[a\langle q@\mathsf{clos}(z, \mathcal{E}, Q), \emptyset, \mathsf{0}\rangle b])$ [S!]
$a = \mathcal{E}(x)$

$(\mathcal{E}, ?y; Q, H[a\langle \mathsf{clos}(z, \mathcal{F}, R), \emptyset, \mathsf{0}\rangle b]) \mapsto (\mathcal{E}', Q, H)$ [S?]
$b = \mathcal{E}(y), \mathcal{E}' = \mathcal{E}\{\mathsf{clos}(z, \mathcal{F}, R)/y\}$

$(\mathcal{E}, \mathsf{call}\ y(w{:}+); Q, H) \mapsto (\mathcal{E}', Q, H[a\langle \mathsf{nil}, \mathcal{F}', R\rangle b])$ [Scall+]
$a, b = \mathsf{new}, \mathcal{E}' = \mathcal{E}\{a/w\}, \mathcal{F}' = \mathcal{F}\{b/z\}$
$\mathsf{clos}(z, \mathcal{F}, R) = \mathcal{E}(y)$

$(\mathcal{E}, \mathsf{call}\ y(w{:}-); Q, H) \mapsto (\mathcal{F}', R, H[a\langle \mathsf{nil}, \mathcal{E}', Q\rangle b])$ [Scall-]
$a, b = \mathsf{new}, \mathcal{E}' = \mathcal{E}\{b/w\}, \mathcal{F}' = \mathcal{F}\{a/z\}$
$\mathsf{clos}(z, \mathcal{F}, R) = \mathcal{E}(y)$

$a\langle q\rangle^s b \triangleq \text{if } (q = \mathsf{nil}) \text{ then } b\langle q, \mathcal{G}, P\rangle a \text{ else } a\langle q, \mathcal{G}, P\rangle b$

Fig. 11: SAM Transition Rules for the complete CLL

$$\frac{enc(\mathcal{E}\{\mathsf{clos!}(y, \mathcal{E}, R)/x\}, P), H) \overset{\mathsf{cut}*}{\Rightarrow} C}{enc(\mathcal{E}, \mathsf{cut!}\ \{y.R \mid !x \mid P\}, H) \overset{\mathsf{cut}*}{\Rightarrow} C}$$

We now update our meta-theoretical results for the complete SAM.

Theorem 6.1 (Soundness). *Let $P \vdash^{\mathsf{B}} \emptyset; \emptyset$.*
If $enc(P) \Rightarrow D \overset{\mathsf{cut}*}{\Rightarrow} C$ then there is Q such that $P \to \cup \equiv Q$ and $C = enc(Q)$.

Theorem 6.2 (Progress). *Let $P \vdash^{\mathsf{B}} \emptyset; \emptyset$ and P live. Then $enc(P) \Rightarrow C$.*

6.1 Concurrent Semantics of Cut and Mix

Intuitively, the execution of mix $P \mid\mid Q$ consists in the parallel execution of (non-interfering) processes P and Q. We may execute $P \mid\mid Q$ by sequentialising P and Q in some arbirary way, and this actually may be useful in some cases.

However, much more interesting is the accommodation in the SAM of interfering concurrency, as required to support full-fledged concurrent languages for session-based programming. First, we evolve the SAM from single threaded to multithreaded, where states now expose a multiset of processes P_i ready for execution by the basic SAM sequential transitions: $(\{P_1, P_2, \ldots, P_n\}, H)$ and introduce an annotated variant pcut of the cut. It has the same $\mathsf{CLL}/\mathsf{CLLB}$ semantics, but to be implemented as a fork construct where P and Q spawn concurrently, their interaction mediated by an atomic *concurrent session record* $x\,\langle q\rangle\,y$. The type system ensuring that concurrent channels may be forwarded only to concurrent channels. We extend the SAM with transition rule for multisets:

$$\frac{(P, H) \Rightarrow (P', H')}{(P \uplus T, H) \Rightarrow (P' \uplus T, H')}\ [\text{Srun}]$$

$$(\mathsf{pcut}\ \{P \mid \overline{x}{:}A\ [q]\ y{:}B \mid Q\} \uplus T, H) \Rightarrow (\{P, Q\} \uplus T\}, H[x\,\langle \mathsf{nil}\rangle\,y]) \quad [\text{SCutp}]$$

$$((P \mid\mid Q) \uplus T, H) \Rightarrow (\{P, Q\} \uplus T\}, H[x\,\langle \mathsf{nil}\rangle\,y]) \quad [\text{SMixp}]$$

Each individual thread executes locally according to the SAM sequential transitions presented before, until an action on a concurrent queue is reached. Concurrent process actions on concurrent queues are atomic, and defined as expected. Positive actions always progress by pushing a value into the queue, while negative actions will either pop off a value from the queue or block, waiting for a value to become available. We illustrate with the rules for $\mathbf{1}$,$\bot$ typed actions.

$$(\mathsf{close}\ x, H[x\,\langle q\rangle\,y]) \Rightarrow (\mathsf{0}, H[x\,\langle q @ \checkmark\rangle\,y]) \qquad [\text{S}\mathbf{1}\text{c}]$$

$$(\mathsf{wait}\ y; P, H[x\langle \checkmark, y\rangle]) \Rightarrow (P, H) \qquad [\text{S}\bot\text{c}]$$

Notice that, as in the case for $\mathsf{wait}\ y; P$ above, any negative action in the thread queue is unable to progress if the corresponding queue is empty. It should be clear how to define transition rules for all other pairs of dual actions. Given an appropriate encoding enc^c of annotated CLLB processes in concurrent SAM states, and as consequence of typing and leveraging the proof scheme for progress in CLLB (Theorem 4.2), we have:

Theorem 6.3 (Soundness-c). *Let $P \vdash^{\mathsf{B}} \emptyset; \emptyset$.*
If $enc^c(P) \Rightarrow D \overset{\mathsf{cut}*}{\Rightarrow} C$ then there is Q such that $P \to \cup \equiv Q$ and $C = enc^c(Q)$.

Theorem 6.4 (Progress-c). *Let $P \vdash^{\mathsf{B}} \emptyset; \emptyset$ and P live. Then $enc_c(P) \mapsto\!\!\!\!\Rightarrow C$.*

The extended SAM executes concurrent session programs, consisting in an arbitrary number of concurrent threads. Each thread deterministically executes sequential code, but can at any moment spawn new concurrent threads. The whole model is expressed in the common language of (classical) linear logic, statically ensuring safety, proper resource usage, termination, and deadlock absence by static typing.

7 Concluding Remarks and Related Work

We introduce the Session Abstract Machine, or SAM, an abstract machine for executing session processes typed by (classical) linear logic CLL, deriving a deterministic, sequential evaluation strategy, where exactly one process is executing at any given point in time. In the SAM, session channels are implemented as single queues with a write and a read endpoint, which are written to, and read by executing processes. Positive actions are non-blocking, giving rise to a degree of asynchrony. However, processes in a session synchronise at polarity inversions, where they alternate execution, according to a fixed co-routining strategy. Despite its specific strategy, the SAM semantics is sound wrt CLL and satisfies the correctness properties of logic-based session type systems. We also present a conservative concurrent extension of the SAM, allowing the degrees of concurrency to be modularly expressed at a fine grain, ranging from fully sequential to fully concurrent execution. Indeed, a practical concern with the SAM design lies in providing a principled foundation for an execution environment for multi-paradigm languages, combining concurrent, imperative and functional programming. The overall SAM design as presented here may be uniformly extended to cover any other polarised language constructs that conservatively extend the PaT paradigm, such as polymorphism, affine types, recursive and co-recursive types, and shared state [56, 61]. We have implemented a SAM-based version [17] of an open-source implementation of CLL [62].

A machine model provides evidence of the algorithmic feasibility of a programming language abstract semantics, and illuminates its operational meaning from certain concrete semantic perspective. Since the seminal work of Landin on the SECD [43], several machines to support the execution of programs for a given programming language have been proposed. The SAM is then proposed herein in this same spirit of Cousineau, Curien and Mauny's Categorical Abstract Machine for the call-by-value λ-calculus [21], Lafont's Linear Abstract Machine for the linear λ-calculus [41], and Krivine's Machine for the call-by-name λ-calculus [40] ; these works explored Curry-Howard correspondences to propose provably correct solutions. In [22], Danvy developed a deconstruction of the SECD based on a sequence of program transformations. The SAM is also derived from Curry-Howard correspondences for linear logic CLL [15, 72], and we also rely on program conversions, via the intermediate buffered language CLLB, as a key proof technique. We believe that the SAM is the first proposal of its kind to tackle the challenges of a process language, while building on several deep properties of its type structure towards a principled design. Among those, focusing [4] and polarisation [44, 32, 56] played an important role to achieve a deterministic sequential reduction strategy for session-based programming, perhaps our main initial motivation. That allows the SAM to naturally and efficiently integrate the execution of sequential and concurrent session behaviours, and suggests effective compilation schemes for mainstream virtual machines or compiler frameworks.

The adoption of session and linear types is clearly increasing in research (e.g., [26, 3, 58, 24, 74, 66, 56, 61]) and general purpose languages (e.g., Haskell [8, 38], Rust [42, 20] Ocaml [35, 52], F# [51], Move [9], among many others), which either require sophisticated encodings of linear typing via type-level computation or forego of some static correctness properties for usability purposes. Such developments typically have as a main focus the realization of the session typing discipline (or of a particular refinement of such typing), with the underlying concurrent execution model often offloaded to existing language infrastructure.

We highlight the work [19], which studies the relationship between synchronous session types and game semantics, which are fundamentally asynchronous. Their work proposes an encoding of synchronous strategies into asynchronous strategies by so-called call-return protocols. While their focus differs significantly from ours, the encoding via asynchrony is reminiscent of our own.

We further note the work [50] which develops a polarized variant of the $\overline{\lambda}\mu\tilde{\mu}$ suitable for sequent calculi like that of linear logic. While we draw upon similar inspirations in the design of the SAM, there are several key distinctions: the work [50] presents $\lambda\mu$-calculi featuring values and substitution of terms for variables (potentially deep within the term structure). Our system, being based on processes calculus, features neither -– there is no term representing the outcome of a computation, since computation is the interactive behavior of processes (cf. game semantics); nor does computation rely on substitution in the same sense. Another significant distinction is that our work materializes a heap-based abstract machine rather than a stack-based machine. Finally, our type and term structure is not itself polarized. Instead, we draw inspiration from focusing insofar as we extract from focusing the insights that drive execution in the SAM.

In future work, we plan to study the semantics of the SAM in terms of games (and categories), along the lines of [19, 21, 41]. We also plan to investigate the ways in which the evaluation strategy of the SAM can be leveraged to develop efficient compilation of fine-grained session-based programming, and its relationship with effect handlers, coroutines and delimited continuations. Linearity plays a key role in programming languages and environments for smart contracts in distributed ledgers [24, 64] manipulating linear resources (assets); it would be interesting to investigate how linear abstract machines like the SAM would provide a basis for certifying resource sensitive computing infrastructures [75, 9].

Data Availability. An implementation of the SAM as a typechecker and interpreter is publicly available [17]. Additional definitions and proofs can be found in the companion extended technical report [16].

Acknowledgments. This work was supported by NOVA LINCS (UIDB/ 04516/ 2020), INESC ID (UIDB/ 50021/ 2020), BIG (Horizon EU 952226 BIG).

References

1. Abramsky, S.: Computational Interpretations of Linear Logic. Theoret. Comput. Sci. **111**(1–2), 3–57 (1993)
2. Abramsky, S., Gay, S.J., Nagarajan, R.: Interaction categories and the foundations of typed concurrent programming. In: NATO ASI DPD. pp. 35–113 (1996)
3. Almeida, B., Mordido, A., Thiemann, P., Vasconcelos, V.T.: Polymorphic lambda calculus with context-free session types. Inf. Comput. **289**(Part), 104948 (2022)
4. Andreoli, J.M.: Logic Programming with Focusing Proofs in Linear Logic. J. Log. Comput. **2**(3), 297–347 (1992)
5. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. **1**(ICFP) (2017)
6. Bellin, G., Scott, P.: On the π-calculus and linear logic. Theoret. Comput. Sci. **135**(1), 11–65 (1994)
7. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models. In: International Workshop on Computer Science Logic. pp. 121–135. Springer (1994)
8. Bernardy, J., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang. **2**(POPL), 5:1–5:29 (2018)
9. Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Russi, D., Sezer, D., Zakian, T., Zhou, R.: Move: A Language with Programmable Resources (2019)
10. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Yang, H. (ed.) Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017. Lecture Notes in Computer Science, vol. 10201, pp. 229–259. Springer (2017)
11. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201. p. 229–259. Springer-Verlag, Berlin, Heidelberg (2017)
12. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: Proceedings of the 22nd European Conference on Programming Languages and Systems. p. 330–349. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013)
13. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory. pp. 222–236. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
14. Caires, L., Pfenning, F., Toninho, B.: Towards concurrent type theory. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. p. 1–12. TLDI '12, Association for Computing Machinery, New York, NY, USA (2012)
15. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016)
16. Caires, L., Toninho, B.: The session abstract machine (extended version) (2024)
17. Caires, L., Toninho, B.: The Session Abstract Machine (Artifact) (2024). https://doi.org/10.5281/zenodo.10459455
18. Cardelli, L.: Typeful Programming. IFIP State-of-the-Art Reports: Formal Description of Programming Concepts pp. 431–507 (1991)
19. Castellan, S., Yoshida, N.: Two sides of the same coin: session types and game semantics: a synchronous side and an asynchronous side. Proc. ACM Program. Lang. **3**(POPL), 27:1–27:29 (2019)

20. Chen, R., Balzer, S., Toninho, B.: Ferrite: A Judgmental Embedding of Session Types in Rust. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022. LIPIcs, vol. 222, pp. 22:1–22:28 (2022)
21. Cousineau, G., Curien, P., Mauny, M.: The Categorical Abstract Machine. Sci. Comput. Program. **8**(2), 173–202 (1987)
22. Danvy, O.: A Rational Deconstruction of Landin's SECD Machine. In: Grelck, C., Huch, F., Michaelson, G., Trinder, P.W. (eds.) Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004. LNCS, vol. 3474, pp. 52–71. Springer (2004)
23. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018. LNCS, vol. 10803, pp. 91–109. Springer (2018)
24. Das, A., Pfenning, F.: Rast: A language for resource-aware session types. Log. Methods Comput. Sci. **18**(1) (2022)
25. DeYoung, H., Caires, L., Pfenning, F., Toninho, B.: Cut reduction in linear logic as asynchronous session-typed communication. In: Computer Science Logic (2012)
26. Franco, J., Vasconcelos, V.T.: A concurrent programming language with refined session types. In: Counsell, S., Núñez, M. (eds.) Software Engineering and Formal Methods - SEFM 2013. LNCS, vol. 8368, pp. 15–28. Springer (2013)
27. Frumin, D., D'Osualdo, E., van den Heuvel, B., Pérez, J.A.: A bunch of sessions: a propositions-as-sessions interpretation of bunched implications in channel-based concurrency. Proc. ACM Program. Lang. **6**(OOPSLA2), 841–869 (2022)
28. Gay, S., Hole, M.: Subtyping for Session Types in the Pi Calculus. Acta Informatica **42**(2-3), 191–225 (2005)
29. Gay, S., Vasconcelos, V.: Linear Type Theory for Asynchronous Session Types. Journal of Functional Programming **20**(1), 19–50 (2010)
30. Girard, J.: A new constructive logic: Classical logic. Math. Struct. Comput. Sci. **1**(3), 255–296 (1991)
31. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR'93. pp. 509–523. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
32. Honda, K., Laurent, O.: An exact correspondence between a typed pi-calculus and polarised proof-nets. Theor. Comput. Sci. **411**(22-24), 2223–2238 (2010)
33. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems. pp. 122–138. Springer (1998)
34. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., et al.: Foundations of Session Types and Behavioural Contracts. ACM Comput. Surv. **49**(1), 3 (2016)
35. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming, ECOOP 2020. LIPIcs, vol. 166, pp. 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
36. Jacobs, J., Balzer, S.: Higher-order leak and deadlock free locks. Proc. ACM Program. Lang. **7**(POPL), 1027–1057 (2023)
37. Klabnik, S., Nichols, C.: The Rust Programming Language (2021)
38. Kokke, W., Dardha, O.: Deadlock-free session types in linear Haskell. In: Hage, J. (ed.) Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell. pp. 1–13. ACM (2021)
39. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: a fully-abstract semantics for classical processes. Proc. ACM Program. Lang. **3**(POPL), 24:1–24:29 (2019)

40. Krivine, J.: A call-by-name Lambda-calculus Machine. High. Order Symb. Comput. **20**(3), 199–207 (2007)
41. Lafont, Y.: The Linear Abstract Machine. Theor. Comput. Sci. **59**, 157–180 (1988)
42. Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing Multiparty Session Types in Rust. In: Coordination Models and Languages Coordination 2020. Lecture Notes in Computer Science, vol. 12134, pp. 127–136. Springer (2020)
43. Landin, P.J.: The Mechanical Evaluation of Expressions. The Computer Journal, Volume 6, Issue 4, January 1964 **6**(4), 308–320 (1964)
44. Laurent, O.: Polarized Proof-Nets: Proof-Nets for LC. In: Girard, J. (ed.) Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99. LNCS, vol. 1581, pp. 213–227. Springer (1999)
45. Lindley, S., Morris, J.G.: Embedding session types in Haskell. In: Mainland, G. (ed.) Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016. pp. 133–145. ACM (2016)
46. Lopes, L.M.B., Silva, F.M.A., Vasconcelos, V.T.: A virtual machine for a process calculus. In: Nadathur, G. (ed.) Principles and Practice of Declarative Programming, International Conference PPDP'99. Lecture Notes in Computer Science, vol. 1702, pp. 244–260. Springer (1999)
47. Milner, R.: Functions as processes. Math. Struct. Comput. Sci. **2**(2), 119–141 (1992)
48. Milner, R.: Elements of interaction: Turing award lecture. Communications of the ACM **36**(1), 78–89 (1993)
49. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
50. Munch-Maccagnoni, G.: Focalisation and classical realisability. In: Grädel, E., Kahle, R. (eds.) Computer Science Logic, 23rd international Workshop, CSL 2009. LNCS, vol. 5771, pp. 409–423. Springer (2009)
51. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in f#. In: Dubach, C., Xue, J. (eds.) Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria. pp. 128–138. ACM (2018)
52. Padovani, L.: A simple library implementation of binary sessions. J. Funct. Program. **27**, e4 (2017)
53. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations and observational equivalences for session-based concurrency. Information and Computation **239**, 254–302 (2014)
54. Pfenning, F., Pruiksma, K.: Relating message passing and shared memory, proof-theoretically. In: Jongmans, S., Lopes, A. (eds.) Coordination Models and Languages - COORDINATION 2023. LNCS, vol. 13908, pp. 3–27. Springer (2023)
55. Pfenning, F.: Structural cut elimination. In: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science. p. 156. LICS '95, IEEE Computer Society, USA (1995)
56. Pfenning, F., Griffith, D.: Polarized Substructural Session Types. In: Proc. of FoSSaCS 2015. LNCS, vol. 9034, pp. 3–22. Springer (2015)
57. Pierce, B.C., Turner, D.N.: Pict: a programming language based on the pi-calculus. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction, Essays in Honour of Robin Milner. pp. 455–494. The MIT Press (2000)
58. Poças, D., Costa, D., Mordido, A., Vasconcelos, V.T.: System f^{μ} $_{\text{ømega}}$ with context-free session types. In: Wies, T. (ed.) Programming Languages and Systems

- 32nd European Symposium on Programming, ESOP 2023. LNCS, vol. 13990, pp. 392–420. Springer (2023)

59. Qian, Z., Kavvos, G., Birkedal, L.: Client-server sessions in linear logic. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–31 (2021)
60. Rocha, P., Caires, L.: Propositions-as-types and Shared State. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–30 (2021)
61. Rocha, P., Caires, L.: Safe session-based concurrency with shared linear state. In: Wies, T. (ed.) Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023. LNCS, vol. 13990, pp. 421–450. Springer (2023)
62. Rocha, P., Caires, L.: Safe session-based concurrency with shared linear state (artifact) (January 2023). https://doi.org/10.5281/zenodo.7506064
63. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press, USA (2001)
64. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.: Safer smart contract programming with Scilla. Proc. ACM Program. Lang. **3**(OOPSLA), 185:1–185:30 (2019)
65. Toninho, B., Caires, L., Pfenning, F.: Functions as Session-Typed Processes. In: FoSSaCS'12. No. 7213 in LNCS (2012)
66. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 350–369. Springer (2013)
67. Toninho, B., Caires, L., Pfenning, F.: A decade of dependent session types. In: Veltri, N., Benton, N., Ghilezan, S. (eds.) PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming. pp. 3:1–3:3. ACM (2021)
68. Toninho, B., Yoshida, N.: On polymorphic sessions and functions: A tale of two (fully abstract) encodings. ACM Trans. Program. Lang. Syst. **43**(2) (Jun 2021)
69. Turner, D.N.: The polymorphic Pi-calculus : theory and implementation. Ph.D. thesis, University of Edinburgh, UK (1996)
70. Vasconcelos, V.T.: Lambda and pi calculi, CAM and SECD machines. J. Funct. Program. **15**(1), 101–127 (2005)
71. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 273–286. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012)
72. Wadler, P.: Propositions as Sessions. Journal of Functional Programming **24**(2-3), 384–418 (2014)
73. Wadler, P.: Propositions as Types. Commun. ACM **58**(12), 75–84 (2015)
74. Willsey, M., Prabhu, R., Pfenning, F.: Design and implementation of concurrent C0. In: Cervesato, I., Fernández, M. (eds.) Proceedings Fourth International Workshop on Linearity, LINEARITY 2016. EPTCS, vol. 238, pp. 73–82 (2016)
75. Wood, G.: Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)

Dependent Types

Trocq: Proof Transfer for Free, With or Without Univalence*

Cyril Cohen[1](✉), Enzo Crance[2,3](✉), and Assia Mahboubi[2,4](✉)

[1] Université Côte d'Azur, Inria, Sophia Antipolis, France
`Cyril.Cohen@inria.fr`

[2] Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, Nantes, France
`{Enzo.Crance,Assia.Mahboubi}@inria.fr`

[3] Mitsubishi Electric R&D Centre Europe, Rennes, France

[4] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Abstract. This article presents Trocq, a new proof transfer framework for dependent type theory. Trocq is based on a novel formulation of type equivalence, used to generalize the univalent parametricity translation. This framework takes care of avoiding dependency on the axiom of univalence when possible, and may be used with more relations than just equivalences. We have implemented a corresponding plugin for the Coq interactive theorem prover, in the Coq-Elpi meta-language.

Keywords: Parametricity, Representation independence, Univalence, Proof assistants, Proof transfer

1 Introduction

Formalizing mathematics provides every object and statement of the mathematical literature with an explicit data structure, in a certain choice of foundational formalism. As one would expect, several such explicit representations are most often needed for a same mathematical concept. Sometimes, these different choices are already made explicit on paper: multivariate polynomials can for instance be represented as lists of coefficient-monomial pairs, *e.g.*, when computing Gröbner bases, but also as univariate polynomials with polynomial coefficients, *e.g.*, for the purpose of projecting algebraic varieties. The conversion between these equivalent data structures however remains implicit on paper, as they code in fact for the same free commutative algebra. In some other cases, implementation details are just ignored on paper, *e.g.*, when a proof involves both reasoning with Peano arithmetic and computing with large integers.

Example 1 (Proof-oriented vs. computation-oriented data structures). The standard library of the Coq interactive theorem prover [32] has two data structures

* This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No.101001995).

for representing natural numbers. Type ℕ is the base-1 number system and the associated elimination principle `ℕ_ind` is the usual recurrence scheme:

```
Inductive ℕ : Type := Oℕ : ℕ | Sℕ (n : ℕ) : ℕ.

ℕ_ind : ∀ P : ℕ → □, P Oℕ → (∀ n : ℕ, P n → P (S n)) → ∀ n : ℕ, P n
```

On the other hand, type `N` provides a binary representation `positive` of non-negative integers, as sequences of bits with a head 1, and is thus better suited for coding efficient arithmetic operations. The successor function `SN : N → N` is no longer a constructor of the type, but can be implemented as a program, via an auxiliary successor function `Spos` for type `positive`.

```
Inductive positive : Type :=
  xI : positive → positive | xO : positive → positive | xH : positive.

Inductive N : Type := ON : N | Npos : positive → N.

Fixpoint Spos (p : positive) : positive := match p with
  | xH ⇒ xO xH | xO p ⇒ xI p | xI p ⇒ xO (Spos p) end.

Definition SN (n : N) := match n with
  | Npos p ⇒ Npos (Spos p) | _ ⇒ Npos xH end.
```

This successor function is useful to implement conversions $\uparrow_{\mathbb{N}}$: `N` → ℕ and $\downarrow_{\mathbb{N}}$: ℕ → `N` between the unary and binary representations. These conversion functions are in fact inverses of each other. The natural recurrence scheme on natural numbers thus *transfers* to type `N`:

```
N_ind : ∀ P : N → □, P ON → (∀ n : N, P n → P (SN n)) → ∀ n : N, P n
```

Incidentally, `N_ind` can be proved from `ℕ_ind` by using only the fact that $\downarrow_{\mathbb{N}}$ is a left inverse of $\uparrow_{\mathbb{N}}$, and the following compatibility lemmas:

$$\downarrow_{\mathbb{N}} \mathtt{O}_{\mathbb{N}} = \mathtt{O}_{\mathtt{N}} \quad \text{and} \quad \forall n : \mathbb{N}, \quad \downarrow_{\mathbb{N}} (\mathtt{S}_{\mathbb{N}}\ n) = \mathtt{S}_{\mathtt{N}}\ (\downarrow_{\mathbb{N}}\ n)$$

Proof transfer issues are not tied to program verification. For instance, the formal study of summation and integration, in basic real analysis, provides a classic example of frustrating bureaucracy.

Example 2 (Extended domains). Given a sequence $(u_n)_{n\in\mathbb{N}}$ of non-negative real numbers, *i.e.*, a function $u : \mathbb{N} \to [0, +\infty[$, u is said to be *summable* when the sequence $(\sum_{k=0}^{n} u_k)_{n\in\mathbb{N}}$ has a finite limit, denoted $\sum u$. Now for two summable sequences u and v, it is easy to see that $u+v$, the sequence obtained by point-wise addition of u and v, is also a summable sequence, and that:

$$\sum(u+v) = \sum u + \sum v \tag{1}$$

As expression $\sum u$ only makes sense when u is a summable sequence, any algebraic operation "under the sum", *e.g.*, rewriting $\sum(u+(v+w))$ into $\sum((w+u)+v)$, *a priori* requires a proof of summability for every rewriting step. In a classical setting, the standard approach rather assigns a default value to the case of an infinite sum, and introduces an extended domain $[0, +\infty]$. Algebraic operations on real numbers, like addition, are extended to the extra $+\infty$ case. Now for a sequence $u : \mathbb{N} \to [0, +\infty]$, the limit $\sum u$ is always defined, as increasing partial sums either converge to a finite limit, or diverge to $+\infty$. The road map is then to first prove that Equation 1 holds for *any* two sequences of *extended* non-negative numbers. The result is then *transferred* to the special case of summable sequences of non-negative numbers. Major libraries of formalized mathematics including Lean's mathlib [1], Isabelle/HOL's Archive of Formal Proofs, coq-interval [20] or Coq's mathcomp-analysis [2], resort to such extended domains and transfer steps, notably for defining measure theory. Yet, as reported by expert users [18], the associated transfer bureaucracy is essentially done manually and thus significantly clutters formal developments in real and complex analysis, probabilities, etc.

Users of interactive theorem provers should be allowed to elude mundane arguments pertaining to proof transfer, as they would on paper, and spare themselves the related bureaucracy. Yet, they still need to convince the proof checker and thus have to provide explicit transfer proofs, albeit ideally automatically generated ones. The present work aims at providing a general method for implementing this nature of automation, for a diverse range of proof transfer problems.

In this paper, we focus on interactive theorem provers based on dependent type theory, such as Coq, Agda [24] or Lean [22]. These proof management systems are genuine functional programming languages, with full-spectrum dependent types, a context in which representation independence meta-theorems can be turned into concrete instruments for achieving program and proof transfer.

Seminal results on the contextual equivalence of distinct implementations of a same abstract interface were obtained for System F, using logical relations [21] and parametricity meta-theorems [26,35]. In the context of type theory, such meta-theorems can be turned into syntactic translations of the type theory of interest into itself, automating this way the generation of the statement and proof of parametricity properties for type families and for programs. Such syntactic relational models can accommodate dependent types [10], inductive types [9] and scale to the Calculus of Inductive Constructions, with an impredicative sort [19].

In particular, the *univalent parametricity* translation [30] leverages the univalence axiom [33] so as to transfer statements using established equivalences of types. This approach crucially removes the need for devising an explicit common interface for the types in relation. In presence of an internalized univalence axiom and of higher-inductive types, the *structure identity principle* provides internal representations of independence results, for more general relations between types than equivalences [5]. This last approach is thus particularly relevant in cubical type theory [12,34]. Indeed, a computational interpretation of the univalence axiom brings computational adequacy to otherwise possibly stuck terms, those resulting from a transfer involving an axiomatized univalence principle.

Yet taming the bureaucracy of proof transfer remains hard in practice for users of Coq, Lean or Agda. Examples 1 and 2 actually illustrate fundamental limitations of the existing approaches:

Univalence is overkill Both univalent parametricity and the structure identity principle can be used to derive the statement and the proof of the induction principle `N_ind` of Example 1, from the elimination scheme of type $\mathbb{N}$. But up to our knowledge, all the existing methods for automating this implication pull in the univalence principle in the proof, although it can be obtained by hand by very elementary means. This limitation is especially unsatisfactory for developers of libraries formalizing classical mathematics, and notably Lean's mathlib. These libraries indeed typically assume a strong form of proof irrelevance, which is incompatible with univalence, and thus with univalent parametricity.

Equivalences are not enough, neither are quotients Univalent parametricity cannot help with Example 2, as type $[0, +\infty[$ is *not equivalent* to its extended version $[0, +\infty]$. In fact, we are not aware of any tool able to automate this proof transfer. In particular, the structure identity principle [5] would not apply as such.

Contributions In short, existing techniques for transferring results from one type to another, *e.g.*, from `ℕ` to `N` or from extended real numbers to real numbers, are either not suitable for dependent types, or too coarse to track the exact amount of data needed in a given proof, and not more. This paper presents three contributions improving this unfortunate state of affairs:

– A parametricity framework *à la carte*, that generalizes the univalent parametricity translation [30], as well as refinements à la CoqEAL [14] and generalized rewriting [28]. Its pivotal ingredient is a variant of Altenkirch and Kaposi's symmetrical presentation of type equivalence [3].
– A conservative subtyping extension of CC_ω [15], used to formulate an inference algorithm for the synthesis of parametricity proofs.
– The implementation of a new parametricity plugin for the Coq interactive theorem prover, using the Coq-Elpi [31] meta-language. This plugin rests on original formal proofs, conducted on top of the HoTT library [8], and is distributed with a collection of application examples.

Outline The rest of this paper is organized as follows. Section 2 introduces proof transfer and recalls the principle, strengths and weaknesses of the univalent parametricity translation. In Section 3, we present a new definition of type equivalence, motivating a hierarchy of structures for relations preserved by parametricity. Section 4 then presents variants of parametricity translations. In Section 5, we discuss a few examples of applications and we conclude in Section 6.

2 Strengths and limits of univalent parametricity

We first clarify the essence of proof transfer in dependent type theory (§ 2.1) and briefly recall a few concepts related to type equivalence and univalence (§ 2.2). We then review and discuss the limits of univalent parametricity (§ 2.3).

2.1 Proof transfer in type theory

We recall the syntax of the Calculus of Constructions, CC_ω, a λ-calculus with dependent function types and a predicative hierarchy of universes, denoted $\square_i$:

$$A, B, M, N ::= \square_i \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$

We omit the typing rules of the calculus, and refer the reader to standard references (*e.g.*, [25,23]). We also use the standard equality type, called propositional equality, as well as dependent pairs, denoted $\Sigma x : A.\, B$. We write $t \equiv u$ the definitional equality between two terms t and u. Interactive theorem provers like Coq, Agda and Lean are based on various extensions of this core, notably with inductive types or with an impredicative sort. When the universe level does not matter, we casually remove the annotation and use notation $\square$.

In this context, proof transfer from type T_1 to type T_2 roughly amounts to *synthesizing* a new type former $W : T_2 \to \square$, *i.e.*, a type parametric in some type T_2, from an initial type former $V : T_1 \to \square$, *i.e.*, a type parametric in some type T_1, so as to ensure that for some given relations $R_T : T_1 \to T_2 \to \square$ and $R_\square : \square \to \square \to \square$, there is a proof w that:

$$\Gamma \vdash w : \forall (t_1 : T_1)(t_2 : T_2), R_T\ t_1\ t_2 \to R_\square (V\ t_1)(W\ t_2)$$

for a suitable context Γ. This setting generalizes as expected to k-ary type formers, and to more pairs of related types. In practice, relation $R_\square$ is often a right-to-left arrow, *i.e.*, $R_\square\ A\ B \triangleq B \to A$, as in this case the proof w substantiates a proof step turning a goal clause $\Gamma \vdash V\ t_1$ into $\Gamma \vdash W\ t_2$.

Phrased as such, this synthesis problem is arguably quite loosely specified. Consider for instance the transfer problem discussed in Example 1. A first possible formalization involves the design of an appropriate common interface structure for types $\mathbb{N}$ and $\mathtt{N}$, for instance by setting both T_1 and T_2 as $\Sigma N : \square. N \times (N \to N)$, and both V and W as: $\lambda X : T_1.\, \Pi P : X.1 \to \square.\, P\ X.2 \to (\Pi n : X.1.\, P\ n \to P\ (X.3\ n)) \to \Pi n : X.1.\, P\ n$, where $X.i$ denotes the i-th item in the dependent tuple X. In this case, relation R_T may characterize isomorphic instances of the structure. Such instances of proof transfer are elegantly addressed in cubical type theories via internal representation independence results [5]. In the context of CC_ω, the hassle of devising explicit structures by hand has been termed the *anticipation* problem [30].

Another option is to consider two different types $T_1 \triangleq \mathbb{N} \times (\mathbb{N} \to \mathbb{N})$ and $T_2 \triangleq \mathrm{N} \times (\mathrm{N} \to \mathrm{N})$ and

$$V' \triangleq \lambda X : T_1.\, \forall P : \mathbb{N} \to \square.\ P\ X.1 \to (\forall n : \mathbb{N}, P\ n \to P(X.2\ n)) \to \forall n : \mathbb{N}, P\ n$$
$$W' \triangleq \lambda X : T_2.\, \forall P : \mathrm{N} \to \square.\ P\ X.1 \to (\forall n : \mathrm{N}, P\ n \to P(X.2\ n)) \to \forall n : \mathrm{N}, P\ n$$

where one would typically expect R_T to be a type equivalence between T_1 and T_2, so as to transport $(V'\ t_1)$ to $(W'\ t_2)$, along this equivalence.

Note that some solutions of given instances of proof transfer problems are in fact too trivial to be of interest. Consider for example the case of a *functional* relation between T_2 and T_1, with $R_T\ t_1\ t_2$ defined as $t_1 = \phi\ t_2$, for some $\phi : T_2 \to T_1$. In this case, the composition $V \circ \phi$ is an obvious candidate for W, but is often uninformative. Indeed, this composition can only propagate structural arguments, blind to the additional mathematical proofs of program equivalences potentially available in the context. For instance, here is a useless variant of W':

$$\begin{aligned} W'' \triangleq \lambda X : T_2.\quad & \forall P : \mathbb{N} \to \square.\ P\ (\uparrow_{\mathbb{N}} X.1) \to \\ & (\forall n : \mathbb{N}, P\ n \to P\,(\uparrow_{\mathbb{N}} (X.2\ (\downarrow_{\mathbb{N}} n)))) \to \forall n : \mathbb{N}, P\ n. \end{aligned}$$

Automation devices dedicated to proof transfer thus typically consist of a meta-program which attempts to compute type former W and proof w by induction on the structure of V, by composing registered canonical pairs of related terms, and the corresponding proofs. These tools differ by the nature of relations they can accommodate, and by the class of type formers they are able to synthesize. For instance, *generalized rewriting* [28], which provides essential support to formalizations based on setoids [7], addresses the case of homogeneous (and reflexive) relations, *i.e.*, when T_1 and T_2 coincide. The CoqEAL library [14] provides another example of such transfer automation tool, geared towards *refinements*, typically from a proof-oriented data-structure to a computation-oriented one. It is thus specialized to heterogeneous, functional relations but restricted to closed, quantifier-free type formers. We now discuss the few transfer methods which can accommodate dependent types and heterogeneous relations.

2.2 Type equivalences, univalence

Let us first focus on the special case of types related by an *equivalence*, and start with a few standard definitions, notations and lemmas. Omitted details can be found in the usual references, like the Homotopy Type Theory book [33]. Two functions $f, g : A \to B$ are *point-wise equal*, denoted $f \doteqdot g$ when their values coincide on all arguments, that is $f \doteqdot g \triangleq \Pi a : A.\, f\ a = g\ a$. For any type A, id_A denotes $\lambda a : A.\, a$, the identity function on A, and we write id when the implicit type A is not ambiguous.

Definition 1 (Type isomorphism, type equivalence). *A function $f : A \to B$ is an* isomorphism*, denoted $\mathsf{IsIso}(f)$, if there exists a function $g : B \to A$ which satisfies the section and retraction properties, i.e., g is respectively a point-wise left and right inverse of f. A function f is an* equivalence*, denoted $\mathsf{IsEquiv}(f)$, when it moreover enjoys a* coherence property*, relating the proofs of the section and retraction properties and ensuring that $\mathsf{IsEquiv}(f)$ is proof-irrelevant.*

Types A and B are equivalent*, denoted $A \simeq B$, when there is an equivalence $f : A \to B$:*

$$A \simeq B \ \triangleq\ \Sigma f : A \to B.\, \mathsf{IsEquiv}(f)$$

Lemma 1. *Any isomorphism $f : A \to B$ is also an equivalence.*

The data of an equivalence $e : A \simeq B$ thus include two *transport functions*, denoted respectively $\uparrow_e : A \to B$ and $\downarrow_e : B \to A$. They can be used for proof transfer from A to B, using $\uparrow_e$ at covariant occurrences, and $\downarrow_e$ at contravariant ones. The *univalence principle* asserts that equivalent types are interchangeable, in the sense that all universes are univalent.

Definition 2 (Univalent universe). *A universe $\mathcal{U}$ is univalent if for any two types A and B in $\mathcal{U}$, the canonical map $A = B \to A \simeq B$ is an equivalence.*

In variants of CC_ω, the *univalence axiom* has no explicit computational content: it just postulates that all universes $\square_i$ are univalent, as for instance in the HoTT library for the Coq interactive theorem prover [8]. Some more recent variants of dependent type theory [12,4] feature a built-in computational univalence principle. They are used to implement experimental interactive theorem provers, such as Cubical Agda [34]. In both cases, the univalence principle provides a powerful proof transfer principle from $\square$ to $\square$, as for any two types A and B such that $A \simeq B$, and any $P : \square \to \square$, we can obtain that $P\ A \simeq P\ B$ as a direct corollary of univalence. Concretely, $P\ B$ is obtained from $P\ A$ by appropriately allocating the transfer functions provided by the equivalence data, a transfer process typically useful in the context of proof engineering [27].

Going back to our example from § 2.1, transferring along an equivalence $\mathbb{N} \simeq \mathtt{N}$ thus produces W'' from V'. Assuming univalence, one may achieve the more informative transport from V' to W', using a method called *univalent parametricity* [30], which we discuss in the next section.

2.3 Parametricity translations

Univalent parametricity strengthens the transfer principle provided by the univalence axiom by combining it with parametricity. In CC_ω, the essence of parametricity, which is to devise a relational interpretation of types, can be turned into an actual syntactic translation, as relations can themselves be modeled as λ-terms in CC_ω. The seminal work of Bernardy, Lasson *et al.* [10,19,9] combine in what we refer to as the *raw parametricity translation*, which essentially defines inductively a logical relation $[\![T]\!]$ for any type T, as described on Figure 1. This presentation uses the standard convention that t' is the term obtained from a term t by replacing every variable x in t with a fresh variable x'. A variable x is translated into a variable x_R, where x_R is a fresh name. Parametricity follows from the associated fundamental theorem, also called abstraction theorem [26]:

Theorem 1. *If $\Gamma \vdash t : T$ then the following hold: $[\![\Gamma]\!] \vdash t : T$, $[\![\Gamma]\!] \vdash t' : T'$ and $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]\ t\ t'$.*

Proof. By structural induction on the typing judgment, see for instance [19].

A key, albeit mundane ingredient of Theorem 1 is the fact that the rules of Figure 1 ensure that:

$$\vdash [\![\square_i]\!] \ :\ [\![\square_{i+1}]\!]\ \square_i\ \square_i \tag{9}$$

This translation precisely generates the statements expected from a parametric type family or program. For instance, the translation of a Π-type, given by

– Context translation:

$$\llbracket \langle\rangle \rrbracket = \langle\rangle \tag{2}$$

$$\llbracket \Gamma, x : A \rrbracket = \llbracket \Gamma \rrbracket, x : A, x' : A', x_R : \llbracket A \rrbracket\ x\ x' \tag{3}$$

– Term translation:

$$\llbracket \Box_i \rrbracket = \lambda A\, A'.\, A \to A' \to \Box_i \tag{4}$$

$$\llbracket x \rrbracket = x_R \tag{5}$$

$$\llbracket A\ B \rrbracket = \llbracket A \rrbracket\ B\ B'\ \llbracket B \rrbracket \tag{6}$$

$$\llbracket \lambda x : A.\, t \rrbracket = \lambda (x : A)(x' : A')(x_R : \llbracket A \rrbracket\ x\ x').\, \llbracket t \rrbracket \tag{7}$$

$$\llbracket \Pi x : A.\, B \rrbracket = \lambda f\, f'.\, \Pi (x : A)(x' : A')(x_R : \llbracket A \rrbracket\ x\ x').\, \llbracket B \rrbracket (f\ x)(f'\ x') \tag{8}$$

Fig. 1: Raw parametricity translation for CC_ω.

Equation 8, is a type of relations on functions that relate those producing related outputs from related inputs. Concrete implementations of this translation are available [19,31]; they generate and prove parametricity properties for type families or for constants, improved induction schemes, etc.

Univalent parametricity follows from the observation that the abstraction theorem still holds when restricting to relations that are in fact (heterogeneous) equivalences. This however requires some care in the translation of universes:

$$\llbracket \Box_i \rrbracket\ A\ B \ \triangleq\ \Sigma (R : A \to B \to \Box_i)(e : A \simeq B).\\ \Pi (a : A)(b : B).\, R\ a\ b \simeq (a =\downarrow_e b) \tag{10}$$

where $\llbracket \cdot \rrbracket$ now refers to the *univalent* parametricity translation, replacing the notation introduced for the raw variant. For any two types A and B, $\llbracket \Box_i \rrbracket\ A\ B$ packages a relation R and an equivalence e such that R is equivalent to the functional relation associated with $\downarrow_e$. Crucially, assuming univalence, $\llbracket \Box_i \rrbracket$ is equivalent to type equivalence, that is, for any two types A and B:

$$\llbracket \Box_i \rrbracket\ A\ B \simeq (A \simeq B).$$

This observation is actually an instance of a more general technique available for constructing syntactic models of type theory [11], based on attaching extra intensional specifications to negative type constructors. In these models, a standard way to recover the abstraction theorem consists of refining the translation into two variants, for any term $T : \Box_i$, that is also a type. The translation of such a T as a *term*, denoted $[T]$, is a dependent pair, which equips a relation with the additional data prescribed by the interpretation $\llbracket \Box_i \rrbracket$ of the universe. The translation $\llbracket T \rrbracket$ of T as a *type* is the relation itself, that is, the projection of the dependent pair $[T]$ onto its first component, denoted $\mathsf{rel}([T])$. We refer to the original article [30, Figure 4] for a complete description of the translation.

We now state the abstraction theorem of the univalent parametricity translation [30], where $\vdash_u$ denotes a typing judgment of CC_ω assuming univalence:

Theorem 2. *If* $\Gamma \vdash t : T$ *then* $[\![\, \Gamma \,]\!] \vdash_u [\, t \,] \;:\; [\![\, T \,]\!]\ t\ t'$.

Note that proving the abstraction theorem 2 involves in particular proving that:

$$\vdash_u [\, \Box_i \,] \;:\; [\![\, \Box_{i+1} \,]\!]\ \Box_i\ \Box_i \quad \text{and} \quad \mathsf{rel}([\, \Box_i \,]) \equiv [\![\, \Box_i \,]\!]. \tag{11}$$

The definition of relation $[\, \Box_i \,]$ relies on the univalence principle in a crucial way, in order to prove that the relation in the universe is equivalent to equality on the universe, *i.e.*, to prove that:

$$\vdash_u \Pi A\, B : \Box_i.\, [\![\, \Box_i \,]\!]\ A\ B \simeq (A = B).$$

Importantly, this univalent parametricity translation can be seamlessly extended so as to also make use of a global context of user-defined equivalences.

Yet because of the interpretation of universes given by Equation 10, univalent parametricity can only automate proof transfer based on *type equivalences*. This is too strong a requirement in many cases, *e.g.*, to deduce properties of natural numbers from that of integers, or more generally for refinement relations. Even in the case of equivalent types, this restriction may be problematic, as Equation 11 may incur unnecessary dependencies on the univalence axiom, as in Example 1.

3 Type equivalence in kit

In this section, we propose (§ 3.1) an equivalent, modular presentation of type equivalence, phrased as a nested sigma type. Then (§ 3.2), we carve a hierarchy of structures on relations out of this dependent tuple, selectively picking pieces. Last, we revisit (§ 3.3) parametricity translations through the lens of this finer grained analysis of the relational interpretations of types.

3.1 Disassembling type equivalence

Let us first observe that the Definition 1, of type equivalence, is quite asymmetrical, although this fact is somehow swept under the rug by the infix $A \simeq B$ notation. First, the data of an equivalence $e : A \simeq B$ privileges the left-to-right direction, as $\uparrow_e$ is directly accessible from e as its first projection, while accessing the right-to-left transport requires an additional projection. Second, the statement of the coherence property, which we eluded in Definition 1, is actually:

$$\Pi a : A.\, \mathsf{ap}_{\uparrow_e}(s\ a) = r \circ \downarrow_e$$

where $\mathsf{ap}_f(t)$ is the term $f\ u = f\ v$, for any identity proof $t : u = v$. This statement uses proofs s and r, respectively of the section and retraction properties of e, but not in a symmetrical way, although swapping them leads to an equivalent definition. This entanglement prevents tracing the respective roles of each direction of transport, left-to-right or right-to-left, during the course of a given univalent parametricity translation. Exercise 4.2 in the HoTT book [33] however suggests a symmetrical definition of type equivalence, via functional relations.

Definition 3. *A relation $R : A \to B \to \square_i$, is* functional *when:*

$$\Pi a : A.\, \mathsf{IsContr}(\Sigma b : B.\, R\ a\ b)$$

where for any type T, $\mathsf{IsContr}(T)$ is the standard contractibility predicate $\Sigma t : T.\, \Pi t' : T.\, t = t'$. This property is denoted $\mathsf{IsFun}(R)$.

We can now obtain an equivalent but symmetrical characterization of type equivalence, as a functional relation whose symmetrization is also functional.

Lemma 2. *For any types $A, B : \square$, type $A \simeq B$ is equivalent to:*

$$\Sigma R : A \to B \to \square.\, \mathsf{IsFun}(R) \times \mathsf{IsFun}(R^{-1})$$

where $R^{-1} : B \to A \to \square$ just swaps the arguments of relation $R : A \to B \to \square$.

We sketch below a proof of this result, left as an exercise in [33]. The essential argument is the following characterization of functional relations:

Lemma 3. *The type of functions is equivalent to the type of functional relations; i.e., for any types $A, B : \square$, we have $(A \to B) \simeq \Sigma R : A \to B \to \square.\, \mathsf{IsFun}(R)$.*

Proof. The proof goes by chaining the following equivalences:

$$\begin{aligned}(\Sigma R : A \to B \to \square.\, \mathsf{IsFun}(R)) \ &\simeq\ (A \to \Sigma P : B \to \square.\, \mathsf{IsContr}(\Sigma b : B.\, P\ b)) \\ &\simeq (A \to B)\end{aligned}$$

Proof (of Lemma 2). The proof goes by chaining the following equivalences, where the type of f is always $A \to B$ and the type of R is $A \to B \to \square$:

$$\begin{array}{rcll}(A \simeq B) & \simeq & \Sigma f : A \to B.\, \mathsf{IsEquiv}(f) & \text{by definition of } (A \simeq B) \\ & \simeq & \Sigma f.\, \Pi b : B.\, \mathsf{IsContr}(\Sigma a. f\ a = b) & \text{standard result in HoTT} \\ & \simeq & \Sigma f.\, \mathsf{IsFun}(\lambda (b : B)(a : A).\, f\ a = b) & \text{by definition of } \mathsf{IsFun}(\cdot) \\ & \simeq & \Sigma\, (\varphi : \Sigma R.\, \mathsf{IsFun}(R))\, .\, \mathsf{IsFun}(\pi_1(\varphi)^{-1}) & \text{by Lemma 3} \\ & \simeq & \Sigma R.\, \mathsf{IsFun}(R) \times \mathsf{IsFun}(R^{-1}) & \text{by associativity of } \Sigma.\end{array}$$

However, the definition of type equivalence provided by Lemma 2 does not expose explicitly the two transfer functions in its data, although this computational content can be extracted via first projections of contractibility proofs. In fact, it is possible to devise a definition of type equivalence which directly provides the two transport functions in its data, while remaining symmetrical. This variant follows from an alternative characterization of functional relations.

Definition 4. *For any types $A, B : \square$, a relation $R : A \to B \to \square$, is a* univalent map*, denoted $\mathsf{IsUmap}(R)$ when there exists a function $m : A \to B$ together with:*

$$\begin{array}{rl} & g_1 : \Pi(a : A)(b : B).\, m\ a = b \to R\ a\ b \\ \textit{and} & g_2 : \Pi(a : A)(b : B).\, R\ a\ b \to m\ a = b \\ \textit{such that} & \Pi(a : A)(b : B).\, (g_1\ a\ b) \circ (g_2\ a\ b) \doteqdot id.\end{array}$$

Now comes the crux lemma of this section.

Lemma 4. *For any types $A, B : \square$ and any relation $R : A \to B \to \square$*

$$\mathsf{IsFun}(R) \simeq \mathsf{IsUmap}(R).$$

Proof. The proof goes by rewording the left hand side, in the following way:

$$\begin{aligned}
&\Pi x.\, \mathsf{IsContr}(R\ x)\\
&\quad \simeq \Pi x.\, \Sigma(r : \Sigma y.\, R\ x\ y).\, \Pi(p : \Sigma y.\, R\ x\ y).\, r = p\\
&\quad \simeq \Pi x.\, \Sigma y.\, \Sigma(r : R\ x\ y).\, \Pi(p : \Sigma y.\, R\ x\ y).\, (y, r) = p\\
&\quad \simeq \Sigma f.\, \Pi x.\, \Sigma(r : R\ x\ (f\ x)).\, \Pi(p : \Sigma y.\, R\ x\ y).\, (f\ x, r) = p\\
&\quad \simeq \Sigma f.\, \Sigma(r : \Pi x.\, R\ x\ (f\ x)).\, \Pi x.\, \Pi(p : \Sigma y.\, R\ x\ y).\, (f\ x, r\ x) = p\\
&\quad \simeq \Sigma f.\, \Sigma r.\, \Pi x.\, \Pi y.\, \Pi(p : R\ x\ y).\, (f\ x, r\ x) = (y, p)\\
&\quad \simeq \Sigma f.\, \Sigma r.\, \Pi x.\, \Pi y.\, \Pi(p : R\ x\ y).\, \Sigma(e : f\ x = y).\, r\ x =_e p\\
&\quad \simeq \Sigma f.\, \Sigma r.\, \Sigma(e : \Pi x.\, \Pi y.\, R\ x\ y \to f\ x = y).\, \Pi x.\, \Pi y.\, \Pi p.\, (r\ x) =_{e\,x\,y\,p} p
\end{aligned}$$

After a suitable reorganization of the sigma types we are left to show that

$$\begin{aligned}
&\Sigma(r : \Pi x.\, \Pi y.\, f\ x = y \to R\ x\ y).\, (e\,x\,y) \circ (r\,x\,y) \doteqdot id\\
\simeq\ &\Sigma(r : \Pi x.\, R\ x\ (f\ x)).\, \Pi x.\, \Pi y.\, \Pi p.\, r\ x =_{e\,x\,y\,p} p
\end{aligned}$$

which proof we do not detail, referring the reader to the supplementary material.

As a direct corollary, we obtain a novel characterization of type equivalence:

Theorem 3. *For any types $A, B : \square_i$, we have:*

$$(A \simeq B) \simeq \boxbox^{\top}\ A\ B$$

where the relation $\boxbox^{\top}\ A\ B$ is defined as:

$$\Sigma R : A \to B \to \square_i.\, \mathsf{IsUmap}(R) \times \mathsf{IsUmap}(R^{-1})$$

The collection of data packed in a term of type $\boxbox^{\top}\ A\ B$ is now symmetrical, as the right-to-left direction of the equivalence based on univalent maps can be obtained from the left-to-right by flipping the relation and swapping the two functionality proofs. If the η-rule for records is verified, symmetry is even *definitionally* involutive.

3.2 Reassembling type equivalence

Definition 4 of univalent maps and the resulting rephrasing of type equivalence suggest introducing a hierarchy of structures for heterogeneous relations, which explains how close a given relation is to type equivalence. In turn, this distance is described in terms of structure available respectively on the left-to-right and right-to-left transport functions.

Definition 5. *For $n, k \in \{0, 1, 2_a, 2_b, 3, 4\}$, and $\alpha = (n, k)$, relation $\boxdot^{\alpha} : \Box \to \Box \to \Box$, is defined as:*

$$\boxdot^{\alpha} \triangleq \lambda(A\ B : \Box).\Sigma(R : A \to B \to \Box).\mathsf{Class}_{\alpha}\ R$$

where the map class $\mathsf{Class}_{\alpha}\ R$ *itself unfolds to a pair type* $(\mathsf{M}_n\ R) \times (\mathsf{M}_k\ R^{-1})$, *with* M_i *defined as:*[5]

$$\begin{aligned}
\mathsf{M}_0\ R &\triangleq .\\
\mathsf{M}_1\ R &\triangleq (A \to B)\\
\mathsf{M}_{2_a}\ R &\triangleq \Sigma m : A \to B.\, G_{2_a}\ m\ R \qquad \textit{with } G_{2_a}\ m\ R \triangleq \Pi a\, b.\, m\ a = b \to R\ a\ b\\
\mathsf{M}_{2_b}\ R &\triangleq \Sigma m : A \to B.\, G_{2_b}\ m\ R \qquad \textit{with } G_{2_b}\ m\ R \triangleq \Pi a\, b.\, R\ a\ b \to m\ a = b\\
\mathsf{M}_3\ R &\triangleq \Sigma m : A \to B.\, (G_{2_a}\ m\ R) \times (G_{2_b}\ m\ R)\\
\mathsf{M}_4\ R &\triangleq \Sigma m : A \to B.\, \Sigma(g_1 : G_{2_a}\ m\ R).\, \Sigma(g_2 : G_{2_b}\ m\ R).\, \Pi a\, b.\\
&\qquad (g_1\ a\ b) \circ (g_2\ a\ b) \doteq id
\end{aligned}$$

For any types A and B, and any $r : \boxdot^{\alpha}\ A\ B$ we use notations $\mathsf{rel}(r)$, $\mathsf{map}(r)$ and $\mathsf{comap}(r)$ to refer respectively to the relation, map of type $A \to B$, map of type $B \to A$, included in the data of r, for a suitable α.

Definition 6. *We denote $\mathcal{A}$ the set $\{0, 1, 2_a, 2_b, 3, 4\}^2$, used to index map classes in Definition 5. This set is partially ordered for the product order defined from the partial order $0 < 1 < 2_* < 3 < 4$ for 2_* either 2_a or 2_b, and with 2_a and 2_b being incomparable.*

Remark 1. Relation $\boxdot^{(4,4)}$ of Definition 5 coincides with the relation $\boxdot^{\top}$ introduced in Theorem 3. Similarly, we denote $\boxdot^{\perp}$ the relation $\boxdot^{(0,0)}$.

Remark 2. Definition 5 is associated with the following dictionary. For r of type:

– $\boxdot^{(1,0)}\ A\ B$, $\mathsf{map}(r)$ is an arbitrary function $f : A \to B$;
– $\boxdot^{(4,0)}\ A\ B$, $\mathsf{rel}(r)$ is a univalent map, in the sense of Definition 4;
– $\boxdot^{(4,2_a)}\ A\ B$, $\mathsf{rel}(r)$ is the graph of a retraction (*i.e.*, a surjective univalent map with an explicit partial left inverse) of type $A \to B$;
– $\boxdot^{(4,2_b)}\ A\ B$, $\mathsf{rel}(r)$ is the graph of a section (*i.e.*, an injective univalent map with explicit partial right inverse) of type $A \to B$;
– $\boxdot^{(4,4)}$, r is an equivalence between A and B;
– $\boxdot^{(3,3)}$, r is an isomorphism between A and B.

Observe that $\boxdot^{(n,m)}\ A\ B$ coincides, up to equivalence, with $\boxdot^{(m,n)}\ B\ A$. Other classes, while not corresponding to a meaningful mathematical definition, may arise in concrete runs of proof transfer: see also Section 4 for explicit examples.

The corresponding lattice to the collection of M_n is implemented as a hierarchy of dependent tuples, more precisely, of record types.

[5] For the sake of readability, we omit implicit arguments, *e.g.*, although M_i has type $\lambda(T_1\ T_2 : \Box).\,(T_1 \to T_2 \to \Box) \to \Box$, we write $\mathsf{M}_n\ R$ for $(\mathsf{M}_n\ A\ B\ R)$.

3.3 Populating the hierarchy of relations

We shall now revisit the parametricity translations of Section 2.3. In particular, combining Theorem 3 with Equation 11, crux of the abstraction theorem for univalent parametricity, ensures the existence of a term $p_{\square_i}$ such that:

$$\vdash_u p_{\square_i} \;:\; \boxdot^{\top}_{i+1}\ \square_i\ \square_i \quad \text{and} \quad \mathsf{rel}(p_{\square_i}) \simeq \boxdot^{\top}_i .$$

Otherwise said, relation $\boxdot^{\top} : \square \to \square \to \square$ can be endowed with a $\boxdot^{\top}$ structure, assuming univalence. Similarly, Equation 9, for the raw parametricity translation, can be read as the fact that relation $\boxdot^{\perp}$ on universes can be endowed with a $\boxdot^{\perp}\ \square\ \square$ structure.

Now the hierarchy of structures introduced by Definition 5 enables a finer grained analysis of the possible relational interpretations of universes. Not only would this put the raw and univalent parametricity translations under the same hood, but it would also allow for generalizing parametricity to a larger class of relations. For this purpose, we generalize the previous observation, on the key ingredient for translating universes: for each $\alpha \in \mathcal{A}$, relation $\boxdot^{\alpha} : \square \to \square \to \square$ may be endowed with several structures from the lattice, and we need to study which ones, depending on α. Otherwise said, we need to identify the pairs $(\alpha, \beta) \in \mathcal{A}^2$ for which it is possible to construct a term $p_{\square}^{\alpha,\beta}$ such that:

$$\vdash_u p_{\square}^{\alpha,\beta} \;:\; \boxdot^{\beta}\ \square\ \square \quad \text{and} \quad \mathsf{rel}(p_{\square}^{\alpha,\beta}) \equiv \boxdot^{\alpha} \tag{12}$$

Note that we aim here at a definitional equality between $\mathsf{rel}(p_{\square}^{\alpha,\beta})$ and $\boxdot^{\alpha}$, rather than at an equivalence. It is easy to see that a term $p_{\square}^{\alpha,\perp}$ exists for any $\alpha \in \mathcal{A}$, as $\boxdot^{\perp}$ requires no structure on the relation. On the other hand, it is not possible to construct a term $p_{\square}^{\perp,\top}$, *i.e.*, to turn an arbitrary relation into a type equivalence.

Definition 7. *We denote $\mathcal{D}_{\square}$ the following subset of $\mathcal{A}^2$:*

$$\mathcal{D}_{\square} = \{(\alpha, \beta) \in \mathcal{A}^2 \mid \alpha = \top \vee \beta \in \{0, 1, 2_a\}^2\}$$

The supplementary material constructs terms $p_{\square}^{\alpha,\beta}$ for every pair $(\alpha, \beta) \in \mathcal{D}_{\square}$, using a meta-program to generate them from a minimal collection of manual definitions. In particular, assuming univalence, it is possible to construct a term $p_{\square}^{\top,\top}$, which can be seen as an analogue of the translation $[\square]$ of univalent parametricity. More generally, the provided terms $p_{\square}^{\alpha,\beta}$ depend on univalence if and only if $\beta \notin \{0, 1, 2_{\mathrm{a}}\}^2$.

The next natural question concerns the possible structures $\boxdot^{\gamma}$ endowing the relational interpretation of a product type $\Pi x : A.\, B$, given relational interpretation for types A and B respectively equipped with structures $\boxdot^{\alpha}$ and $\boxdot^{\beta}$.

Otherwise said, we need to identify the triples $(\alpha, \beta, \gamma) \in \mathcal{A}^3$ for which it is possible to construct a term p_{Π}^{γ} such that the following statements both hold:

$$\frac{\Gamma \vdash A_R \;:\; \boxdot^{\alpha}\ A\ A' \qquad \Gamma,\ x : A,\ x' : A',\ x_R : A_R\ x\ x' \vdash B_R \;:\; \boxdot^{\beta}\ B\ B'}{\Gamma \vdash p_{\Pi}^{\gamma}\ A_R\ B_R \;:\; \boxdot^{\gamma}\ (\Pi x : A.\, B)\ (\Pi x' : A'.\, B')}$$

$\mathsf{rel}(p_\Pi^\gamma\ A_R\ B_R) \equiv \lambda f.\lambda f'.\Pi(x : A)(x' : A')(x_R : \mathsf{rel}(A_R)\ x\ x').\ \mathsf{rel}(B_R)\ (fx)\ (f'x')$

The corresponding collection of triples can actually be described as a function $\mathcal{D}_\Pi : \mathcal{A} \to \mathcal{A}^2$, such that $\mathcal{D}_\Pi(\gamma) = (\alpha, \beta)$ provides the *minimal* requirements on the structures associated with A and B, with respect to the partial order on $\mathcal{A}^2$. The supplementary material provides a corresponding collection of terms p_Π^γ for each $\gamma \in \mathcal{A}$, as well as all the associated weakenings. Once again, these definitions are generated by a meta-program. Observe in particular that by symmetry, $p_\Pi^{(m,n)}$ can be obtained from $p_\Pi^{(m,0)}$ and $p_\Pi^{(n,0)}$ by swapping the latter and glueing it to the former. Therefore, the values of p_Π^γ and $\mathcal{D}_\Pi(\gamma)$ are completely determined by those of $p_\Pi^{(m,0)}$ and $\mathcal{D}_\Pi(m, 0)$. In particular, for any $(m, n) \in \mathcal{A}$:

$$\mathcal{D}_\Pi(m, n) = ((m_A, n_A), (m_B, n_B))$$

where $m_A, n_A, m_B, n_B \in \mathcal{A}$ are such that $\mathcal{D}_\Pi(m, 0) = ((0, n_A), (m_B, 0))$ and $\mathcal{D}_\Pi(n, 0) = ((0, m_A), (n_B, 0))$. We sum up in Figure 2 the values of $\mathcal{D}_\Pi(m, 0)$.

m	$\mathcal{D}_\Pi(m,0)_1$	$\mathcal{D}_\Pi(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,2_\mathrm{a})$	$(1,0)$
2_a	$(0,4)$	$(2_\mathrm{a},0)$
2_b	$(0,2_\mathrm{a})$	$(2_\mathrm{b},0)$
3	$(0,4)$	$(3,0)$
4	$(0,4)$	$(4,0)$

m	$\mathcal{D}_\to(m,0)_1$	$\mathcal{D}_\to(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,1)$	$(1,0)$
2_a	$(0,2_\mathrm{b})$	$(2_\mathrm{a},0)$
2_b	$(0,2_\mathrm{a})$	$(2_\mathrm{b},0)$
3	$(0,3)$	$(3,0)$
4	$(0,4)$	$(4,0)$

Fig. 2: Minimal dependencies for product and arrow types

Note that in the case of a non-dependent product, constructing $p_\to^\gamma$ requires less structure on the domain A of an arrow type $A \to B$, which motivates the introduction of function $\mathcal{D}_\to(\gamma)$. Using the combinator for dependent products to interpret an arrow type, albeit correct, potentially pulls in unnecessary structure (and axiom) requirements. The supplementary material includes a construction of terms $p_\to^\gamma$ for any $\gamma \in \mathcal{A}$.

The two tables in Figure 2 show how requirements on levels stay the same on the right hand side of both Π and $\to$, stay the same up to symmetries (exchange of variance and of 2_a and 2_b) on the left hand side of a $\to$ and increase on the left hand side of a Π. This elegant arithmetic justifies our hierarchy of relations.

4 A calculus for proof transfer

This section introduces Trocq, a framework for proof transfer designed as a generalization of parametricity translations, so as to allow for interpreting types as instances of the structures introduced in Section 3.2. We adopt a sequent style presentation, which fits closely the type system of CC_ω, while explaining in a consistent way the transformations of terms and contexts. This choice of presentation departs from the standard literature about parametricity in pure type systems. Yet, it brings the presentation closer to actual implementations, whose necessary management of parametricity contexts is swept under the rug by notational conventions (*e.g.*, the primes of Section 2.3).

For this purpose, we successively introduce four calculi, of increasing sophistication. We start (§ 4.1) with introducing this sequent style presentation by rephrasing the raw parametricity translation, and the univalent parametricity one (§ 4.2). We then introduce CC_ω^+ (§ 4.3), a Calculus of Constructions with annotations on sorts and subtyping, before defining (§ 4.4) the TROCQ calculus.

4.1 Raw parametricity sequents

We introduce *parametricity contexts*, under the form of a list of triples packaging two variables x and x' together with a third one x_R. The latter x_R is a *witness* (a proof) that x and x' are related:

$$\Xi ::= \varepsilon \mid \Xi,\ x \sim x' \because x_R$$

We write $(x, x', x_R) \in \Xi$ when $\Xi = \Xi',\ x \sim x' \because x_R,\ \Xi''$ for some Ξ' and Ξ''.

We denote $\mathrm{Var}(\Xi)$ the sequence of variables related in a parametricity context Ξ, with multiplicities:

$$\mathrm{Var}(\varepsilon) = \varepsilon \qquad \mathrm{Var}(\Xi,\ x \sim x' \because x_R) = \mathrm{Var}(\Xi), x, x', x_R$$

A parametricity context Ξ is *well-formed*, written $\Xi \vdash$, if the sequence $\mathrm{Var}(\Xi)$ is duplicate-free. In this case, we use the notation $\Xi(x) = (x', x_R)$ as a synonym of $(x, x', x_R) \in \Xi$.

A *parametricity judgment* relates a parametricity context Ξ and three terms M, M', M_R of CC_ω. Parametricity judgments, denoted as:

$$\Xi \vdash M \sim M' \because M_R,$$

are defined by rules of Figure 3 and read *in context Ξ, term M translates to the term M', because M_R.*

Lemma 5. *The relation associating a term M with pairs (M', M_R) such that $\Xi \vdash M \sim M' \because M_R$ holds, with Ξ a well-formed parametricity context, is* functional. *More precisely, for any well-formed parametricity context Ξ:*

$$\forall M, M', N', M_R, N_R, \quad \Xi \vdash M \sim M' \because M_R \quad \wedge \quad \Xi \vdash M \sim N' \because N_R \\ \implies \quad (M', M_R) = (N', N_R)$$

Proof. Immediate by induction on the syntax of M.

This presentation of parametricity thus provides an alternative definition of translation $[\![\cdot]\!]$ from Figure 1, and accounts for the prime-based notational convention used in the latter.

$$\frac{}{\Xi \vdash \Box_i \sim \Box_i \because \lambda(A\,B : \Box_i).\,A \to B \to \Box_i}\ (\textsc{ParamSort})$$

$$\frac{(x, x', x_R) \in \Xi \qquad \Xi \vdash}{\Xi \vdash x \sim x' \because x_R}\ (\textsc{ParamVar})$$

$$\frac{\Xi \vdash M \sim M' \because M_R \quad \Xi \vdash N \sim N' \because N_R}{\Xi \vdash M\ N \sim M'\ N' \because M_R\ N\ N'\ N_R}\ (\textsc{ParamApp})$$

$$\frac{\Xi, x \sim x' \because x_R \vdash M \sim M' \because M_R}{\Xi \vdash \lambda x : A.\,M \sim \lambda x' : A'.\,M' \because \lambda x\,x'\,x_R.\,M_R}\ (\textsc{ParamLam})$$

$$\frac{\begin{array}{c} x, x' \notin \mathrm{Var}(\Xi) \\ \Xi \vdash A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash B \sim B' \because B_R \end{array}}{\Xi \vdash \Pi x : A.\,B \sim \Pi x' : A'.\,B' \because \lambda f\,g.\,\Pi x\,x'\,x_R.\,B_R\ (f\ x)\ (g\ x')}\ (\textsc{ParamPi})$$

Fig. 3: Param: sequent-style binary parametricity translation

Definition 8. *A parametricity context Ξ is* admissible *for a well-formed typing context Γ, denoted $\Gamma \rhd \Xi$, when Ξ and Γ are well-formed as a parametricity context and Γ provides coherent type annotations for all terms in Ξ, that is, for any variables x, x', x_R such that $\Xi(x) = (x', x_R)$, and for any terms A' and A_R:*

$$\Xi \vdash \Gamma(x) \sim A' \because A_R \quad \Longrightarrow \quad \Gamma(x') = A' \ \wedge\ \Gamma(x_R) \equiv A_R\ x\ x'$$

We can now state and prove an abstraction theorem:

Theorem 4 (Abstraction theorem).

$$\frac{\Gamma \rhd \Xi \qquad \Gamma \vdash M : A \qquad \Xi \vdash M \sim M' \because M_R \qquad \Xi \vdash A \sim A' \because A_R}{\Gamma \vdash M' : A' \qquad \textit{and} \qquad \Gamma \vdash M_R : A_R\ M\ M'}$$

Proof. By induction on the derivation of $\Xi \vdash M \sim M' \because M_R$.

4.2 Univalent parametricity sequents

We now propose in Figure 4 a rephrased version of the univalent parametricity translation [30], using the same sequent style and replacing the translation of universes with the equivalent relation $\boxdot^{\top}$. Parametricity judgments are denoted:

$$\Xi \vdash_u M \sim M' \because M_R$$

where Ξ is a parametricity context and M, M', and M_R are terms of CC_ω. The u index is a reminder that typing judgments $\Gamma \vdash_u M : A$ involved in the associated abstraction theorem assume the univalence axiom.

We can now rephrase the abstraction theorem for univalent parametricity.

$$\frac{}{\Xi \vdash_u \Box_i \sim \Box_i \because p_{\Box_i}^{\top,\top}}\ (\text{UParamSort}) \qquad \frac{(x, x', x_R) \in \Xi \qquad \Xi \vdash}{\Xi \vdash_u x \sim x' \because x_R}\ (\text{UParamVar})$$

$$\frac{\Xi \vdash_u M \sim M' \because M_R \qquad \Xi \vdash_u N \sim N' \because N_R}{\Xi \vdash_u M\ N \sim M'\ N' \because M_R\ N\ N'\ N_R}\ (\text{UParamApp})$$

$$\frac{\Xi \vdash_u A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash_u M \sim M' \because M_R}{\Xi \vdash_u \lambda x : A.\, M \sim \lambda x' : A'.\, M' \because \lambda x\, x'\, x_R.\, M_R}\ (\text{UParamLam})$$

$$\frac{\Xi \vdash_u A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash_u B \sim B' \because B_R}{\Xi \vdash_u \Pi x : A.\, B \sim \Pi x' : A'.\, B' \because p_{\Pi}^{\top}\ A_R\ B_R}\ (\text{UParamPi})$$

Fig. 4: UParam: univalent parametricity rules

Theorem 5 (Univalent abstraction theorem).

$$\frac{\Gamma \rhd \Xi \qquad \Gamma \vdash M : A \qquad \Xi \vdash_u M \sim M' \because M_R \qquad \Xi \vdash_u A \sim A' \because A_R}{\Gamma \vdash M' : A' \qquad \textit{and} \qquad \Xi \vdash_u M_R : \textit{rel}(A_R)\ M\ M'}$$

Proof. By induction on the derivation of $\Xi \vdash_u M \sim M' \because M_R$.

Remark 3. In Theorem 5, $\mathsf{rel}(A_R)$ is a term of type $A \to A' \to \Box$. Indeed:

$$\frac{\Gamma \vdash A : \Box_i \qquad \Xi \vdash_u A \sim A' \because A_R \qquad \Gamma \rhd \Xi}{\Gamma \vdash_u A_R : \mathsf{rel}(p_{\Box_i}^{\top,\top})\ A\ A'}$$

entails A_R has type

$$\begin{aligned}\mathsf{rel}(p_{\Box_i}^{\top,\top})\ A\ A' &\equiv \boxdot^{\top}\ A\ A' \\ &\equiv \Sigma R : A \to A' \to \Box.\, \mathsf{IsUmap}(R) \times \mathsf{IsUmap}(R^{-1}).\end{aligned}$$

4.3 Annotated type theory

We are now ready to generalize the relational interpretation of types provided by the univalent parametricity translation, so as to allow for interpreting sorts with instances of weaker structures than equivalence. For this purpose, we introduce a variant CC_ω^+ of CC_ω where each universe is annotated with a label indicating the structure available on its relational interpretation. Recall from Section 3.2 that we have used annotations $\alpha \in \mathcal{A}$ to identify the different structures of the lattice disassembling type equivalence: these are the labels annotating sorts of CC_ω^+, so that if A has type $\Box^\alpha$, then the associated relation A_R has type $\boxdot^\alpha\ A\ A'$. The syntax of CC_ω^+ is thus:

$$M, N, A, B \in \mathcal{T}_{CC_\omega^+} ::= \Box_i^\alpha \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$
$$\alpha \in \mathcal{A} = \{0, 1, 2_{\text{a}}, 2_{\text{b}}, 3, 4\}^2 \qquad i \in \mathbb{N}$$

Before completing the actual formal definition of the Trocq proof transfer framework, let us informally illustrate how these annotations shall drive the interpretation of terms, and in particular, of a dependent product $\Pi x : A.\, B$. In this case, before translating B, three terms representing the bound variable x, its translation x', and the parametricity witness x_R are added to the context. The type of x_R is $\mathsf{rel}(A_R)\ x\ x'$ where A_R is the parametricity witness relating A to its translation A'. The role of annotation α on the sort typing type A is thus to to govern the amount of information available in witness x_R, by determining the type of A_R. This intent is reflected in the typing rules of CC_ω^+ , which rely on the definition of the loci $\mathcal{D}_\Box$, $\mathcal{D}_\rightarrow$ and $\mathcal{D}_\Pi$, introduced in §3.3.

Contexts are defined as usual, but typing terms in CC_ω^+ requires defining a *subtyping* relation $\preccurlyeq$, defined by the rules of Figure 5. The typing rules of CC_ω^+ are available in Figure 6 and follow standard presentations [6]. The $\equiv$ relation in the (SubConv) rule is the *conversion* relation, defined as the closure of α-equivalence and β-reduction. The two types of judgment in CC_ω^+ are thus:

$$\Gamma \vdash_+ A \preccurlyeq B \quad \text{and} \quad \Gamma \vdash_+ M : A$$

where M, A and B are terms in CC_ω^+ and Γ is a context in CC_ω^+.

$$\frac{\Gamma \vdash_+ A : K \qquad \Gamma \vdash_+ B : K \qquad A \equiv B}{\Gamma \vdash_+ A \preccurlyeq B}\ (\textsc{SubConv}) \qquad \frac{\alpha \geq \beta \qquad i \leq j}{\Gamma \vdash_+ \Box_i^\alpha \preccurlyeq \Box_j^\beta}\ (\textsc{SubSort})$$

$$\frac{\Gamma \vdash_+ M'\ N : K \qquad \Gamma \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ M\ N \preccurlyeq M'\ N}\ (\textsc{SubApp})$$

$$\frac{\Gamma, x : A \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ \lambda x : A.\, M \preccurlyeq \lambda x : A.\, M'}\ (\textsc{SubLam})$$

$$\frac{\Gamma \vdash_+ \Pi x : A.\, B : \Box_i \qquad \Gamma \vdash_+ A' \preccurlyeq A \qquad \Gamma, x : A' \vdash_+ B \preccurlyeq B'}{\Gamma \vdash_+ \Pi x : A.\, B \preccurlyeq \Pi x : A'.\, B'}\ (\textsc{SubPi})$$

$$K ::= \Box_i \mid \Pi x : A.\, K$$

Fig. 5: Subtyping rules for CC_ω^+

Due to space constraints, we omit the direct proof that CC_ω^+ is a conservative extension over CC_ω. It goes by defining an erasure function for terms $|\cdot|^- : \mathcal{T}_{CC_\omega^+} \rightarrow \mathcal{T}_{CC_\omega}$ and the associated erasure function for contexts.

4.4 The Trocq calculus

The final stage of the announced generalization consists in building an analogue to the parametricity translations available in pure type systems, but for the annotated type theory of § 4.3. This analogue is geared towards proof transfer, as discussed in § 2.1, and therefore designed to *synthesize* the output of the translation from its input, rather than to *check* that certain pairs of terms are

$$\frac{\Gamma \vdash_+ M : A \qquad \Gamma \vdash_+ A \preccurlyeq B}{\Gamma \vdash_+ M : B}\ (\textsc{Conv}^+) \qquad \frac{(\alpha,\beta) \in \mathcal{D}_\square}{\Gamma \vdash_+ \square_i^\alpha : \square_{i+1}^\beta}\ (\textsc{Sort}^+)$$

$$\frac{(x,A) \in \Gamma \qquad \Gamma \vdash_+}{\Gamma \vdash_+ x : A}\ (\textsc{Var}^+) \qquad \frac{\Gamma \vdash_+ A : \square_i \qquad x \notin \mathrm{Var}(\Gamma)}{\Gamma,\ x : A \vdash_+}\ (\textsc{Context}^+)$$

$$\frac{\Gamma \vdash_+ M : \Pi x : A.\, B \qquad \Gamma \vdash_+ N : A}{\Gamma \vdash_+ M\ N : B[x := N]}\ (\textsc{App}^+) \qquad \frac{\Gamma, x : A \vdash_+ M : B}{\Gamma \vdash_+ \lambda x : A.\, M : \Pi x : A.\, B}\ (\textsc{Lam}^+)$$

$$\frac{\Gamma \vdash_+ A : \square_i^\alpha \qquad \Gamma \vdash_+ B : \square_i^\beta \qquad \mathcal{D}_\to(\gamma) = (\alpha,\beta)}{\Gamma \vdash_+ A \to B : \square_i^\gamma}\ (\textsc{Arrow}^+)$$

$$\frac{\Gamma \vdash_+ A : \square_i^\alpha \qquad \Gamma, x : A \vdash_+ B : \square_i^\beta \qquad \mathcal{D}_\Pi(\gamma) = (\alpha,\beta)}{\Gamma \vdash_+ \Pi x : A.\, B : \square_i^\gamma}\ (\textsc{Pi}^+)$$

Fig. 6: Typing rules for CC_ω^+

in relation. However, splitting up the interpretation of universes into a lattice of possible relation structures means that the source term of the translation is not enough to characterize the desired output: the translation needs to be informed with some extra information about the expected outcome of the translation. In the Trocq calculus, this extra information is a type of CC_ω^+.

We thus define Trocq *contexts* as lists of quadruples:

$$\Delta ::= \varepsilon \mid \Delta,\ x \ @\ A \sim x' \because x_R \quad \text{where } A \in \mathcal{T}_{CC_\omega^+},$$

and introduce a conversion function γ from Trocq contexts to CC_ω^+ contexts:

$$\begin{aligned} \gamma(\varepsilon) &= \varepsilon \\ \gamma(\Delta,\ x \ @\ A \sim x' \because x_R) &= \gamma(\Delta),\ x : A \end{aligned}$$

Now, a Trocq judgment is a 4-ary relation of the form $\Delta \vdash_t M \ @\ A \sim M' \because M_R$, which is read *in context Δ, term M of annotated type A translates to term M', because M_R* and M_R is called a parametricity witness. Trocq judgments are defined by the rules of Figure 7. This definition involves a weakening function for parametricity witnesses, defined as follows.

Definition 9. *For all $p, q \in \{0, 1, 2_a, 2_b, 3, 4\}$, such that $p \geq q$, we define map $\downarrow_q^p$: $\mathsf{M}_p \to \mathsf{M}_q$, which forgets the fields from class M_p that are not in M_q.*

For all $\alpha, \beta \in \mathcal{A}$, such that $\alpha \geq \beta$, function $\downdownarrows_\beta^\alpha$: $\boxdot^\alpha\ A\ B \to \boxdot^\beta\ A\ B$ is defined by:

$$\downdownarrows_{(p,q)}^{(m,n)}\ \langle R, M^\to, M^\leftarrow \rangle := \langle R, \downarrow_p^m M^\to, \downarrow_q^n M^\leftarrow \rangle.$$

The weakening function on parametricity witnesses is defined on Figure 8 by extending function $\downdownarrows_\beta^\alpha$ to all relevant pairs of types of CC_ω^+ , i.e., $\Downarrow_U^T$ is defined for $T, U \in \mathcal{T}_{CC_\omega^+}$ as soon as $T \preccurlyeq U$.

$$\frac{(\alpha,\beta)\in\mathcal{D}_\square}{\varDelta\vdash_t \square_i^\alpha \mathrel{@} \square_{i+1}^\beta \sim \square_i^\alpha \because p_{\square_i}^{\alpha,\beta}}\ (\textsc{TrocqSort})$$

$$\frac{(x,A,x',x_R)\in\varDelta \qquad \gamma(\varDelta)\vdash_+}{\varDelta\vdash_t x \mathrel{@} A \sim x' \because x_R}\ (\textsc{TrocqVar})$$

$$\frac{\varDelta\vdash_t M \mathrel{@} \varPi x:A.\,B \sim M' \because M_R \qquad \varDelta\vdash_t N \mathrel{@} A \sim N' \because N_R}{\varDelta\vdash_t M\ N \mathrel{@} B[x:=N] \sim M'\ N' \because M_R\ N\ N'\ N_R}\ (\textsc{TrocqApp})$$

$$\frac{\begin{array}{c}\varDelta\vdash_t A \mathrel{@} \square_i^\alpha \sim A' \because A_R\\ \varDelta, x \mathrel{@} A \sim x' \because x_R \vdash_t M \mathrel{@} B \sim M' \because M_R\end{array}}{\varDelta\vdash_t \lambda x:A.\,M \mathrel{@} \varPi x:A.\,B \sim \lambda x':A'.\,M' \because \lambda x\,x'\,x_R.\,M_R}\ (\textsc{TrocqLam})$$

$$\frac{\begin{array}{c}(\alpha,\beta)=\mathcal{D}_\rightarrow(\delta)\\ \varDelta\vdash_t A \mathrel{@} \square_i^\alpha \sim A' \because A_R \qquad \varDelta\vdash_t B \mathrel{@} \square_i^\beta \sim B' \because B_R\end{array}}{\varDelta\vdash_t A\rightarrow B \mathrel{@} \square_i^\delta \sim A'\rightarrow B' \because p_\rightarrow^\delta\ A_R\ B_R}\ (\textsc{TrocqArrow})$$

$$\frac{\begin{array}{c}(\alpha,\beta)=\mathcal{D}_\varPi(\delta) \qquad \varDelta\vdash_t A \mathrel{@} \square_i^\alpha \sim A' \because A_R\\ \varDelta, x \mathrel{@} A \sim x' \because x_R \vdash_t B \mathrel{@} \square_i^\beta \sim B' \because B_R\end{array}}{\varDelta\vdash_t \varPi x:A.\,B \mathrel{@} \square_i^\delta \sim \varPi x':A'.\,B' \because p_\varPi^\delta\ A_R\ B_R}\ (\textsc{TrocqPi})$$

$$\frac{\varDelta\vdash_t M \mathrel{@} A \sim M' \because M_R \qquad \gamma(\varDelta)\vdash_+ A\preccurlyeq B}{\varDelta\vdash_t M \mathrel{@} B \sim M' \because \Downarrow_B^A M_R}\ (\textsc{TrocqConv})$$

Fig. 7: Trocq rules

An abstraction theorem relates Trocq judgments and typing in CC_ω^+ .

Theorem 6 (Trocq abstraction theorem).

$$\frac{\begin{array}{c}\gamma(\varDelta)\vdash_+ \qquad \gamma(\varDelta)\vdash_+ M:A\\ \varDelta\vdash_t M \mathrel{@} A \sim M' \because M_R \qquad \varDelta\vdash_t A \mathrel{@} \square_i^\alpha \sim A' \because A_R\end{array}}{\gamma(\varDelta)\vdash_+ M':A' \qquad \textit{and} \qquad \gamma(\varDelta)\vdash_+ M_R : \textit{rel}(A_R)\ M\ M'}$$

Proof. By induction on derivation $\varDelta\vdash_t M \mathrel{@} A \sim M' \because M_R$.

Note that type A in the typing hypothesis $\gamma(\varDelta)\vdash_+ M:A$ of the abstraction theorem is exactly the extra information passed to the translation. The latter can thus also be seen as an inference algorithm, which infers annotations for the output of the translation from that of the input.

Remark 4. Since by definition of $p_\square^{\alpha,\beta}$ (Equation 12), we have $\vdash_t \square^\alpha \mathrel{@} \square^\beta \sim \square^\alpha \because p_\square^{\alpha,\beta}$, by applying Theorem 6 with $\gamma(\varDelta)\vdash_+ A:\square^\alpha$, we get:

$$\frac{\gamma(\varDelta)\vdash_+ A:\square^\alpha \qquad \varDelta\vdash_t A \mathrel{@} \square^\alpha \sim A' \because A_R}{\gamma(\varDelta)\vdash_+ A_R : \mathsf{rel}(p_\square^{\alpha,\beta})\ A\ A'}.$$

$$\Downarrow_{\square_i^{\alpha'}}^{\square_i^{\alpha}} t_R := \downdownarrows_{\alpha'}^{\alpha} t_R \qquad \Downarrow_{A'\ M'}^{A\ M} N_R := \Downarrow_{A'}^{A} M\ M'\ N_R$$

$$\Downarrow_{\lambda x:A'.\,B'}^{\lambda x:A.\,B} M\ M'\ N_R := \Downarrow_{B'[x:=M']}^{B[x:=M]} N_R$$

$$\Downarrow_{\Pi x:A'.\,B'}^{\Pi x:A.\,B} M_R := \lambda x\ x'\ x_R.\ \Downarrow_{B'}^{B} \left(M_R\ x\ x'\ (\Downarrow_{A}^{A'}\ x_R)\right) \qquad \Downarrow_{A'}^{A} M_R := M_R$$

Fig. 8: Weakening of parametricity witnesses

Now by the same definition, for any $\beta \in \mathcal{A}$, $\mathsf{rel}(p_{\square}^{\alpha,\beta}) = \boxdot^{\alpha}$, hence $\gamma(\Delta) \vdash A_R : \boxdot^{\alpha}\ A\ A'$, as expected by the type annotation $A : \square^{\alpha}$ in the premise of the rule.

Remark 5. By applying the Remark 4 with $\vdash_{+} \square^{\alpha} : \square^{\beta}$, we indeed obtain that $\vdash_{+} p_{\square}^{\alpha,\beta} : \boxdot^{\beta}\ \square^{\alpha}\ \square^{\alpha}$ as expected, provided that $(\alpha, \beta) \in \mathcal{D}_{\square}$.

4.5 Constants

Concrete applications require extending Trocq with constants. Constants are similar to variables, except that they are stored in a global context instead of a typing context. A crucial difference though is that a constant may be assigned several different annotated types in CC_{ω}^{+} .

Consider for example, a constant `list`, standing for the type of polymorphic lists. As `list` A is the type of lists with elements of type A, it can be annotated with type $\square^{\alpha} \to \square^{\alpha}$ for any $\alpha \in \mathcal{A}$.

Every constant declared in the global environment has an associated collection of possible annotated types $T_c \subset \mathcal{T}_{CC_{\omega}^{+}}$. We require that all the annotated types of a same constant share the same erasure in CC_{ω}, *i.e.*, $\forall c, \forall A, \forall B,\ A, B \in T_c \Rightarrow |\,A\,|^{-} = |\,B\,|^{-}$. For example, $T_{\texttt{list}} = \{\square^{\alpha} \to \square^{\alpha} \mid \alpha \in \mathcal{A}\}$.

In addition, we provide translations $\mathcal{D}_c(A)$ for each possible annotated type A of each constant c in the global context. For example, $\mathcal{D}_{\texttt{list}}(\square^{(1,0)} \to \square^{(1,0)})$ is equal to (`list`, $\lambda A\, A'\, A_R$. (`List.All2` A_R, `List.map` (`map` A_R))), where relation `List.All2` A_R relates lists of the same length, whose elements are pair-wise related via A_R, `List.map` is the usual map function on lists and `map` $A_R : A \to A'$ extracts the *map* projection of the record A_R of type $\boxdot^{(1,0)}\ A\ A' \equiv \Sigma R.\ A \to A'$. Part of these translations can be generated automatically by weakening.

We describe in Figure 9 the additional rules for constants in CC_{ω}^{+} and Trocq. Note that for an input term featuring constants, an unfortunate choice of annotation may lead to a stuck translation.

$$\frac{c \in \mathcal{C} \qquad A \in T_c}{\Gamma \vdash c : A}\ (\text{Const}^{+}) \qquad\qquad \frac{\mathcal{D}_c(A) = (c', c_R)}{\Delta \vdash c\ @\ A \sim c'\ \because\ c_R}\ (\text{TrocqConst})$$

Fig. 9: Additional constant rules for CC_{ω}^{+} and Trocq

We describe in Figure 9 the additional rules for constants in CC_{ω}^{+} and Trocq. Note that for an input term featuring constants, an unfortunate choice of annotation may lead to a stuck translation.

5 Implementation and applications

The Trocq plugin [13] turns the material presented in Section 4 into an actual tactic, called `trocq`, for automating proof transfer in Coq. This tactic synthesizes a new goal from the current one, as well as a proof that the former implies the latter. User-defined relations between constants, registered via specific vernacular commands, inform this synthesis. The core of the plugin implements each rule of the Trocq calculus in the Elpi meta-programming language [17,31], on top of Coq libraries formalizing the contents of Section 3. In the logic programming paradigm of Elpi, each rule of Figure 7 translates gracefully into a corresponding λProlog predicate, making the corresponding source code very close to the presentation of §4.4. However, the Trocq plugin also implements a much less straightforward annotation inference algorithm, so as to hide the management of sort annotations to Coq users completely. This section illustrates the usage of the `trocq` tactic on various concrete examples.

5.1 Isomorphisms

Bitvectors (code). Here are two possible encodings of bitvectors in Coq:

```
bounded_nat (k : nat) := {n : nat & n < pow 2 k}. (* n < 2^k *)
bitvector (k : nat) := Vector.t Bool k. (* size k vectors of booleans *)
```

We can prove that these representations are equivalent by combining two proofs by transitivity: the proof that `bounded_nat k` is related to `bitvector k` for a given `k`, and the proof that `Vector.t` is related to itself. We also make use of the equivalence relations `natR` and `boolR`, which respectively relate type `nat` and `Bool` with themselves:

```
Rk : ∀ (k : nat), Param44.Rel (bounded_nat k) (bitvector k)
vecR : ∀ (A A' : Type) (AR : Param44.Rel A A') (k k' : nat)
  (kR : natR k k'), Param44.Rel (Vector.t A k) (Vector.t A' k')
(* equivalence between types (bounded_nat k) and (bitvector k') *)
bvR : ∀ (k k' : nat) (kR : natR k k'),
  Param44.Rel (bounded_nat k) (bitvector k')
(* informing Trocq with these equivalences *)
Trocq Use vecR natR boolR bvR.
```

Now, suppose we would like to transfer the following result from the bounded natural numbers to the vector-based encoding:

```
∀ (k : nat) (v : bounded_nat k) (i : nat) (b : Bool), i < k ->
  get (set v i b) i = b
```

As this goal involves `get` and `set` operations on bitvectors, and the order and equality relations on type `nat`, we inform Trocq with the associated operations `getv` and `setv` on the vector encoding. E.g., for `get` and `getv`, we prove:

```
getR : ∀ (k k' : nat) (kR : natR k k')
  (v : bounded_nat k) (v' : bitvector k') (vR : bvR k k' kR v v')
  (n n' : nat) (nR : natR n n'), boolR (get v n) (getv v' n')
```

We can now use proof transfer from bitvectors to bounded natural numbers:

```
Trocq Use eqR ltR. (* where eq and lt are translated to themselves *)
Trocq Use getR setR.

Lemma setBitGetSame : ∀ (k : nat) (v : bitvector k),
∀ (i : nat) (b : Bool), i < k -> getv (setv v i b) i = b.
Proof. trocq. exact setBitGetSame'. (* same lemma, on bitvector *) Qed.
```

Induction principle on integers. (code). Recall that the problem from Example 1 is to obtain the following elimination scheme, from that available on type $\mathbb{N}$:

```
N_ind : ∀ P : N → □, P 0ℕ → (∀ n : N, P n → P (Sℕ n)) → ∀ n : N, P n
```

We first inform Trocq that `N` and `ℕ` are isomorphic, by providing proofs that the two conversions $\uparrow_{\mathbb{N}}$: N $\to$ $\mathbb{N}$ and $\downarrow_{\mathbb{N}}$: $\mathbb{N}$ $\to$ N are mutual inverses. Using lemma `Iso.toParam`, we can deduce an equivalence `Param44.Rel N ℕ`, *i.e.*, $\boxdot^{(4,4)}$ N $\mathbb{N}$. We also prove and register that zeros and successors are related:

```
Definition NR : Param44.Rel N ℕ := ...
Lemma 0R : rel NR 0ℕ 0ℕ.
Lemma SR : ∀ m n, rel NR m n → rel NR (Sℕ m) (Sℕ n).
Trocq Use NR 0R SR.
```

Trocq is now able to generate, prove and apply the desired implication:

```
Lemma N_ind : ∀ P : N → □, P 0ℕ → (∀ n : N, P n → P (Sℕ n)) →
  ∀ n : N, P n.
Proof.
  trocq. (* in the goal, N, 0ℕ, Sℕ have been replaced by ℕ, 0ℕ, Sℕ *)
  exact nat_rect.
Qed.
```

Inspecting this proof confirms that only information up to level $(2_a, 3)$ has been extracted from the equivalence proof $\texttt{N}_R$. It is thus possible to run the exact proof transfer, but with a weaker relation, as illustrated in the code for an abstract type I with a zero and a successor constants, and a retraction $\mathbb{N} \to I$.

5.2 Sections, retractions

Modular arithmetic (code). A typical application of modular arithmetic is to show that some statement on $\mathbb{Z}$ can be reduced to statments on $\mathbb{Z}/p\mathbb{Z}$ Let us show how Trocq can synthesize and prove the following implication:

```
Lemma IntRedModZp : (forall (m n p : Zmodp), (m = n * n)%Zmodp -> m = 0)
  -> forall (m n p : int), (m = n * n)%int -> (m == 0)%int.
Proof. intro Hyp. trocq; simpl. now apply Hyp. Qed.
```

where scope `%Zmodp` is for the usual product and zero on type `Zmodp`, for $\mathbb{Z}/p\mathbb{Z}$, scope `%int` for those on type `int`, for $\mathbb{Z}$, and `==` is an equality test modulo p on type `int`. Observe that the implication deduces a lemma on $\mathbb{Z}$ *from* its modular analogues. Type `Zmodp` and `int` are obviously not equivalent, but a *retraction* is actually enough for this proof transfer. We have:

```
modp : int -> Zmodp
reprp : Zmodp -> int
reprpK : ∀ (x : Zmodp), modp (reprp x) = x
Rp : Param42a.Rel int Zmodp
```

where `Rp`, (a proof that $\boxdot^{(4,2_\mathrm{a})}\ \mathbb{Z}\ \ \mathbb{Z}/p\mathbb{Z}$), is obtained from `reprpK` via lemma `SplitSurj.toParam`. Proving lemma `IntRedModZp` by `trocq` now just requires relating the respective zeroes, multiplications, and equalities of the two types:

```
R0 : Rp 0%int 0%Zmodp.
Rmul : ∀ (m : int) (x : Zmodp) (xR : Rp m x)
   (n : int) (y : Zmodp) (yR : Rp n y), Rp (m * n)%int (x * y)%Zmodp.
Reqmodp : ∀ (m : int) (x : Zmodp), Rp m x ->
  ∀ (n : int) (y : Zmodp), Rp n y -> Param01.Rel (m == n) (x = y).
Trocq Use Rp Rmul R0 Reqmodp. (* informing Trocq with these relations *)
```

where `Param01.Rel P Q` (`Param01.Rel` is the Coq name for $\boxdot^{(0,1)}$) is `Q -> P`. Note that by definition of the relation given by `Rp`, lemma `Rmul` amounts to:

```
  ∀ (m n : int), modp (m * n)%int = (modp m * modp n)%Zmodp.
```

Summable sequences. (code). Example 2 involves two instances of subtypes: type $\overline{\mathbb{R}}_{\geq 0}$ extends a type $\mathbb{R}_{\geq 0}$ of positive real numbers with an abstract element and type `summable` is for provably convergent sequences of positive real numbers:

```
Inductive R̄≥0 : Type := Fin : ℝ≥0 → R̄≥0 | Inf : R̄≥0.
Definition seqℝ≥0 := nat → ℝ≥0. Definition seqR̄≥0 := nat → R̄≥0.
Record summable := {to_seq :> seqℝ≥0; _ : isSummable to_seq}.
```

Type $\overline{\mathbb{R}}_{\geq 0}$ and $\mathbb{R}_{\geq 0}$ are related at level $(4, 2_\mathrm{b})$: *e.g.*, `truncate : R̄≥0 -> ℝ≥0` provides a partial inverse to the `Fin` injection by sending the extra `Inf` to zero. Types `summable` and `seqR̄≥0` are also related at level $(4, 2_\mathrm{b})$, via the relation:

```
Definition Rrseq (u : summable) (v : seqR̄≥0) : Type := seq_extend u = v.
```

where `seq_extend` transforms a summable sequence into a sequence of extended positive reals in the obvious way. Now `ΣR̄≥0 u : R̄≥0` is the sum of a sequence `u : seq`$_{\overline{\mathbb{R}}_{\geq 0}}$ of extended positive reals, and we also define the sum of a sequence of positive reals, as a positive real, again by defaulting infinite sums to zero. For the purpose of the example, we only do so for summable sequences:

```
Definition Σℝ≥0 (u : summable) : ℝ≥0 := truncate (Σℝ̄≥0 (seq_extend u)).
```

These two notions of sums are related via `Rrseq`, and so are the respective additions of positive (resp. extended positive) reals and the respective pointwise additions of sequences. Once Trocq is informed of these relations, the tactic is able to transfer the statement from the much easier variant on extended reals:

```
(* relating type ℝ≥0 and ℝ̄≥0 and their respective equalities *)
Trocq Use Param01_paths Param42b_nnR.
(* relating sequence types, sums, addition, addition of sequences *)
Trocq Use Param4a_rseq R_sum_xnnR R_add_xnnR seq_nnR_add.

Lemma sum_xnnR_add : ∀ (u v : ℝ̄≥0), Σℝ̄≥0 (u + v) = Σℝ̄≥0 u + Σℝ̄≥0 v.
Proof.(...) Qed. (* easy, as no convergence proof is needed *)

Lemma sum_nnR_add : ∀ (u v : ℝ≥0), Σℝ≥0 (u + v) = Σℝ≥0 u + Σℝ≥0 v.
Proof. trocq; exact sum_xnnR_add. Qed.
```

5.3 Polymorphic, dependent types

Polymorphic parameters (code). Suppose we want to transfer a goal involving lists along an equivalence between the types of the values contained in the lists. We first prove that the `list` type former is equivalent to itself, and register this fact:

```
listR : ∀ A A' (AR : Param44.Rel A A'), Param44.Rel (list A) (list A')
Trocq Use listR.
```

We also need to relate with themselves all operations on type `list` involved in the goal, including constructors, and to register these facts, before Trocq is able to transfer any goal, *e.g.*, about `list N` to its analogue on `list` N.

Note that lemma `listR` requires an *equivalence* between its parameters. If this does not hold, as in the case of type `int` and `Zmodp` from Section 5.1, the translation is stuck: weakening does not apply here. In order to avoid stuck translation, we need several versions of `listR` to cover all cases. For instance, the following lemma is required for proof transfers from `list Zmodp` to `list int`.

```
listR2a4 : ∀ A A' (AR : Param2a4.Rel A A'),
  Param2a4.Rel (list A) (list A').
```

Dependent and polymorphic types (code). Fixed-size vectors can be represented by iterated tuples, an alternative to the inductive type `Vector.t`, from Coq's standard library, as follows.

```
Definition tuple (A : Type) : nat -> Type := fix F n :=
  match n with O => Unit | S n' => F n' * A end.
```

On the following mockup example, Trocq transfers a lemma on `Vector.t` to its analogue on `tuple`, about a function `head : ∀ A n, tuple A (S n) -> A`, and a function `const : ∀ A, A -> ∀ n, tuple A n` creating a constant vector, and simultaneously refines integers into the integers modulo p from Section 5.1:

```
Lemma head_cst (n : nat) (i : int): Vector.hd (Vector.const i (S n)) = i.
Proof. destruct n; simpl; reflexivity. Qed. (* easy proof *)

Lemma head_cst' : ∀ (n : nat) (z : Zmodp), head (const z (S n)) = z.
Proof. trocq. exact head_const. Qed.
```

This automated proof only requires proving (and registering) that `head` and `const` are related to their analogue `Vector.hd` and `Vector.const`, from Coq's standard library. Note that the proof uses the equivalence between `Vector.t` and `tuple` but only requires a retraction between parameter types.

6 Conclusion

The Trocq framework can be seen as a generalization of the univalent parametricity translation [30]. It allows for weaker relations than equivalence, thanks to a fine-grained control of the data propagated by the translation. This analysis is enabled by skolemizing the usual symmetrical presentation of equivalence, so as to expose the data, and by introducing a hierarchy of algebraic structures for relations. This scrutiny allows in particular to get rid of the univalence axiom for a larger class of equivalence proofs [29], and to deal with refinement relations for arbitrary terms, unlike the CoqEAL library [14]. Altenkirch and Kaposi already proposed a symmetrical, skolemized phrasing of type equivalence [3], but for different purposes. In particular, they did not study the resulting hierarchy of structures. Definition 4 however slightly differs from theirs: by reducing the amount of transport involved, it eases formal proofs significantly in practice, both in the internal library of Trocq and for end-users of the tactic.

The concrete output of this work is a plugin [13] that consists of about 3000 l. of original Coq proofs and 1200 l. of meta-programming, in the Elpi meta-language, excluding white lines and comments. This plugin goes beyond the state of the art in two ways. First, it demonstrates that a single implementation of this parametricity framework covers the core features of several existing other tactics, for refinements [14,16], generalized rewriting [28], and proof transfer [30]. Second, it addresses use cases, such as Example 2, that are beyond the skills of any existing tool in any proof assistant based on type theory. The prototype plugin arguably needs an improved user interface so as to reach the maturity of some of the aforementioned existing tactics. It would also benefit from an automated generation of equivalence proofs, such as Pumpkin Pi [27].

Acknowledgments. The authors would like to thank András Kovács, Kenji Maillard, Enrico Tassi, Quentin Vermande, Théo Winterhalter, and anonymous reviewers, whose comments greatly helped us improve this article.

Disclosure of Interests. None.

References

1. The lean mathematical library. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020. pp. 367–381. ACM (2020). https://doi.org/10.1145/3372885.3373824, https://doi.org/10.1145/3372885.3373824
2. Affeldt, R., Cohen, C.: Measure construction by extension in dependent type theory with application to integration (2023), accepted for publication in JAR
3. Altenkirch, T., Kaposi, A.: Towards a cubical type theory without an interval. In: TYPES. LIPIcs, vol. 69, pp. 3:1–3:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
4. Angiuli, C., Brunerie, G., Coquand, T., Harper, R., (Favonia), K.H., Licata, D.R.: Syntax and models of cartesian cubical type theory. Math. Struct. Comput. Sci. **31**(4), 424–468 (2021). https://doi.org/10.1017/S0960129521000347, https://doi.org/10.1017/S0960129521000347
5. Angiuli, C., Cavallo, E., Mörtberg, A., Zeuner, M.: Internalizing representation independence with univalence. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021). https://doi.org/10.1145/3434293, https://doi.org/10.1145/3434293
6. Aspinall, D., Compagnoni, A.B.: Subtyping dependent types. Theor. Comput. Sci. **266**(1-2), 273–309 (2001). https://doi.org/10.1016/S0304-3975(00)00175-4, https://doi.org/10.1016/S0304-3975(00)00175-4
7. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. J. Funct. Program. **13**(2), 261–293 (2003). https://doi.org/10.1017/S0956796802004501, https://doi.org/10.1017/S0956796802004501
8. Bauer, A., Gross, J., Lumsdaine, P.L., Shulman, M., Sozeau, M., Spitters, B.: The hott library: a formalization of homotopy type theory in coq. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017. pp. 164–172. ACM (2017). https://doi.org/10.1145/3018610.3018615, https://doi.org/10.1145/3018610.3018615
9. Bernardy, J., Jansson, P., Paterson, R.: Proofs for free - parametricity for dependent types. J. Funct. Program. **22**(2), 107–152 (2012). https://doi.org/10.1017/S0956796812000056, https://doi.org/10.1017/S0956796812000056
10. Bernardy, J., Lasson, M.: Realizability and parametricity in pure type systems. In: Hofmann, M. (ed.) Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6604, pp. 108–122. Springer (2011). https://doi.org/10.1007/978-3-642-19805-2_8, https://doi.org/10.1007/978-3-642-19805-2_8
11. Boulier, S., Pédrot, P., Tabareau, N.: The next 700 syntactical models of type theory. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017. pp. 182–194. ACM (2017). https://doi.org/10.1145/3018610.3018620, https://doi.org/10.1145/3018610.3018620
12. Cohen, C., Coquand, T., Huber, S., Mörtberg, A.: Cubical type theory: A constructive interpretation of the univalence axiom. FLAP **4**(10), 3127–3170 (2017), http://collegepublications.co.uk/ifcolog/?00019

13. Cohen, C., Crance, E., Mahboubi, A.: coq-community/trocq: Trocq 0.1.5 (Jan 2024). https://doi.org/10.5281/zenodo.10563382, https://doi.org/10.5281/zenodo.10563382
14. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: Gonthier, G., Norrish, M. (eds.) Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8307, pp. 147–162. Springer (2013). https://doi.org/10.1007/978-3-319-03545-1_10, https://doi.org/10.1007/978-3-319-03545-1_10
15. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988). https://doi.org/10.1016/0890-5401(88)90005-3, https://doi.org/10.1016/0890-5401(88)90005-3
16. Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to computational algebra in coq. In: Beringer, L., Felty, A. (eds.) Interactive Theorem Proving. pp. 83–98. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
17. Dunchev, C., Guidi, F., Coen, C.S., Tassi, E.: ELPI: fast, embeddable, λprolog interpreter. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9450, pp. 460–468. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_32, https://doi.org/10.1007/978-3-662-48899-7_32
18. Gouëzel, S.: Vitali-carathéodory theorem in mathlib. https://leanprover-community.github.io/mathlib_docs/measure_theory/integral/vitali_caratheodory.html (2021)
19. Keller, C., Lasson, M.: Parametricity in an impredicative sort. In: Cégielski, P., Durand, A. (eds.) Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France. LIPIcs, vol. 16, pp. 381–395. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012). https://doi.org/10.4230/LIPIcs.CSL.2012.381, https://doi.org/10.4230/LIPIcs.CSL.2012.381
20. Martin-Dorel, É., Melquiond, G.: Proving tight bounds on univariate expressions with elementary functions in coq. J. Autom. Reason. **57**(3), 187–217 (2016). https://doi.org/10.1007/s10817-015-9350-4, https://doi.org/10.1007/s10817-015-9350-4
21. Mitchell, J.C.: Representation independence and data abstraction. In: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986. pp. 263–276. ACM Press (1986). https://doi.org/10.1145/512644.512669, https://doi.org/10.1145/512644.512669
22. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37, https://doi.org/10.1007/978-3-030-79876-5_37
23. Nederpelt, R., Geuvers, H.: Type Theory and Formal Proof: An Introduction. Cambridge University Press (2014). https://doi.org/10.1017/CBO9781139567725
24. Norell, U.: Dependently typed programming in agda. In: Koopman, P.W.M., Plasmeijer, R., Swierstra, S.D. (eds.) Advanced Functional Programming, 6th

International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures. Lecture Notes in Computer Science, vol. 5832, pp. 230–266. Springer (2008). https://doi.org/10.1007/978-3-642-04652-0_5, https://doi.org/10.1007/978-3-642-04652-0_5

25. Paulin-Mohring, C.: Introduction to the Calculus of Inductive Constructions. In: Paleo, B.W., Delahaye, D. (eds.) All about Proofs, Proofs for All, Studies in Logic (Mathematical logic and foundations), vol. 55. College Publications (Jan 2015), https://inria.hal.science/hal-01094195
26. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Mason, R.E.A. (ed.) Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983. pp. 513–523. North-Holland/IFIP (1983)
27. Ringer, T., Porter, R., Yazdani, N., Leo, J., Grossman, D.: Proof repair across type equivalences. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 112–127. ACM (2021). https://doi.org/10.1145/3453483.3454033, https://doi.org/10.1145/3453483.3454033
28. Sozeau, M.: A new look at generalized rewriting in type theory. J. Formaliz. Reason. **2**(1), 41–62 (2009). https://doi.org/10.6092/issn.1972-5787/1574, https://doi.org/10.6092/issn.1972-5787/1574
29. Tabareau, N., Tanter, É., Sozeau, M.: Equivalences for free: univalent parametricity for effective transport. Proceedings of the ACM on Programming Languages **2**(ICFP), 1–29 (2018)
30. Tabareau, N., Tanter, É., Sozeau, M.: The marriage of univalence and parametricity. Journal of the ACM (JACM) **68**(1), 1–44 (2021)
31. Tassi, E.: Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. In: ITP 2019 - 10th International Conference on Interactive Theorem Proving. Portland, United States (Sep 2019). https://doi.org/10.4230/LIPIcs.CVIT.2016.23, https://inria.hal.science/hal-01897468
32. The Coq Development Team: The coq proof assistant (Sep 2022). https://doi.org/10.5281/zenodo.7313584, https://doi.org/10.5281/zenodo.7313584
33. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)
34. Vezzosi, A., Mörtberg, A., Abel, A.: Cubical agda: a dependently typed programming language with univalence and higher inductive types. Proc. ACM Program. Lang. **3**(ICFP), 87:1–87:29 (2019). https://doi.org/10.1145/3341691, https://doi.org/10.1145/3341691
35. Wadler, P.: Theorems for free! In: Stoy, J.E. (ed.) Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989. pp. 347–359. ACM (1989). https://doi.org/10.1145/99370.99404, https://doi.org/10.1145/99370.99404

Artifact Report: Trocq: Proof Transfer for Free, With or Without Univalence

Cyril Cohen[1], Enzo Crance[2,3], and Assia Mahboubi[2,4](✉)

[1] Université Côte d'Azur, Inria, Biot, France
Cyril.Cohen@inria.fr
[2] Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, Nantes, France
{Enzo.Crance,Assia.Mahboubi}@inria.fr
[3] Mitsubishi Electric R&D Centre Europe, Rennes, France
[4] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Trocq [5] is both the name of a calculus, describing a parametricity framework, and of a Coq plugin [6] that provides tactics for performing representation changes in goals, as well as vernacular commands for specifying the expected translations. More precisely, from an initial goal of type `G`, the `trocq` tactic simultaneously computes using the Trocq calculus [5] a translation `G'` and a justification `w : G' -> G`. If successful, the user is thus left with proving `G'`.

The plugin orchestrates this double synthesis, by assembling existing building blocks known to the tactic, in the course of a linear traversal of the input term `G`. These building blocks are of two natures. First, the actual rules of the parametricity framework [5] govern the synthesis rule attached to each term construction of CC_ω. The other nature of building blocks is the collection of registered pairs of user-defined constants. These pairs come equipped with a witness of their relatedness at some level, a data registered via the `Trocq Use` command. When the linear traversal of the input term hits a constant, it queries the database registering these user-defined relations, looking for the corresponding constant and witness to be used in the synthesis.

The Trocq plugin is implemented in Elpi [9]: a dialect of λProlog which can be used as a meta-language for Coq, through the Coq-Elpi [20] plugin. The latter encodes Coq terms in higher-order abstract syntax (HOAS) which provides native support for bound variables, complemented by a comprehensive API (typechecking, elaboration, interacting with the global environment, etc).

1 Example

Let us translate the induction principle associated with type `nat`, the unary representation of ℕ, to type `N`, the binary one. Types `nat` and `N` are equivalent and we use the `Trocq Use` command to register such pairs of related types:

```
Definition RN : (N <=> nat)%P := Iso.toParamSym N.of_nat_iso.
Trocq Use RN. (* registering a pair of related types *)
```

Proof `RN` coerces to a relation of type `N -> nat -> Type`, and we also register proofs that it relates the respective zero and successor constants of these types:

```
Definition RN0 : RN 0%N 0%nat. Proof. done. Qed.
Definition RNS m n : RN m n -> RN (N.succ m) (S n). Proof. by case. Qed.
Trocq Use RN0. Trocq Use RNS. (* registering related constants *)
```

We can now use tactic `trocq` to prove a useful induction principle on type `N`:

```
Lemma N_Srec : forall (P : N -> Type), P 0%N ->
    (forall n, P n -> P (N.succ n)) -> forall n, P n.
Proof. trocq. (* replaces N by nat in the goal *) exact nat_rect. Qed.
```

Inspecting the proof term actually reveals that univalence was not needed in the proof of `N_Srec`. The `example` directory of the artifact provides more examples, for weaker relations than equivalences, and beyond representation independence.

2 Architecture of the plugin

A Trocq parametricity sequent $\Delta \vdash_t M \,@\, A \sim M' \because M_R$ expresses that *terms M and M' are related at type A with witness M_R* in context Δ. Unlike standard, unequivocal parametricity translations, each construct of CC_ω gives rise to a *family* of possible synthesis rules, indexed by annotations on M and A.

Encoding CC_ω^+. To implement the annotation calculus CC_ω^+, we just annotate Coq's sort `Type` with a pair (n, m) using *convertible* synonyms `(PType n m)`, where `PType := fun (_ _ : label) => Type`. The two thrown-away arguments code for the annotation. In the course of the synthesis, arguments of certain occurrences of `PType` are left as holes and filled by a constraint solving algorithm.

Synthesis. The logic programming paradigm on which Elpi is based, is ideal to implement algorithms expressed as inference rules, as each rule can be associated to an instance of a predicate. The linear traversal of the input term at the core of the Trocq plugin is operated by the predicate `param`, of arity 4, where `param X T X' XR` stands for the parametricity sequent $\Delta \vdash_t x \,@\, T \sim x' \because x_R$ for a certain context Δ. In this sequent, x and T are input values (initially, the source goal and the annotated sort $\square^{(0,1)}$), and the synthesized term x' and witness x_R are outputs. Each construct of CC_ω leads to *one* instance of the predicate. As an example, let us inspect the instance of the `param` predicate for dependent products, which implements the rule TrocqPi of the Trocq calculus. For the sake of readability, we removed lines related to logs, pretty-printing, and fresh universe instance generation. The head of the predicate is:

```
param (prod N A B) (app [pglobal (const PType) _, M1, M2]) Prod' ProdR :-
  param.db.ptype PType, !,
  cstr.univ-link C M1 M2,
```

which matches an input term $\Pi x : A.\, B$ and our Coq encoding of its annotated type $\square^{(M_1,M_2)}$. Then, following the hypotheses in the inference rule, the predicate computes the prescribed annotation $(C_A, C_B) = \mathcal{D}_\Pi(C)$, and does two recursive calls on A and B with classes C_A and C_B:

```
cstr.dep-pi C CA CB,
cstr.univ-link CA M1A M2A,
param A (app [pglobal (const PType) _, M1A, M2A]) A' AR,
cstr.univ-link CB M1B M2B,
TB = app [pglobal (const PType) _, M1B, M2B],
@annot-pi-decl N A a\ pi a' aR\ param.store a A a' aR =>
  param (B a) TB (B' a') (BR a a' aR),
```

The last step (omitted in the above snippet) is to build the output proof $p_\Pi^C\ A_R\ B_R$. As the axioms (univalence, functional extensionality) that might be involved in some proofs are not assumed globally, they are used as an additional argument albeit only in the building blocks that require them. Therefore, we check whether the requested rule requires the addition of an axiom to the list of arguments (in the case of the dependent product, function extensionality). If this axiom is not present in the context at the time of calling this part of the code, the tactic rightfully fails, because the translation is impossible.

Exploiting symmetries. Trocq provides several distinct rules per language construct (such as Π) and per relation structure among the 36 items in the hierarchy: for a same construct, these rules differ by the annotations required on the input of the rule, and by the structure of the relation relating the input term and the synthesized one. For each such rule, a Coq function provides the corresponding rule building block. Making the most of symmetries, the 495 rule building blocks are generated by meta-programming from only 9 manually defined ones.

Handling of constants. Finally, the traversal of the input term collects *constraints* on the annotations, as multiple valid solutions might exists: for instance, an implication might be obtained from weakening an equivalence. The algorithm strives to minimize the requirements on the user-defined building blocks, which also amounts to minimizing the dependency on axioms. This inference procedure is formalized as a finite domain constraint solving problem, and implemented using *Constraint Handling Rules* (CHR) language [10], as available in Elpi.

3 Related work

In the context of type theory, Barthe and Pons [3] already noticed that the computational content of type isomorphisms can serve proof transfer. The first implementation report of a tool based on this idea appeared soon after [16]. Implemented in a meta-language and based on proof rewriting, this heuristic translation produced a candidate proof term from an existing proof term, with no formal guarantee, not even that of being well-typed. Generalized rewriting [17], which generalizes setoid rewriting to preorders, is also a variant of proof transfer, albeit within the same type. As such, it allows in particular rewriting under binders. The restriction to homogeneous relations however excludes more general instances of proof transfer, *e.g.*, , datatype representation change and quasi-PERs (QPER, or zig-zag complete relations) [13], essentially heterogeneous.

The other proof transfer methods we are aware of all address the case of heterogeneous relations. Incidentally, they can thus also be used for the homogeneous case, and thus for generalized rewriting, although this special case is seldom emphasized. The Coq Effective Algebra Library (CoqEAL) [8,7] and the Isabelle/HOL transfer package [14,11,12,15], pioneered the use of parametricity-based methods for proof transfer, motivated by the refinement of proof-oriented data-structures to computation-oriented counterparts. Together with a subsequent generalization of the CoqEAL approach [21], these tools address the case of a transfer between a subtype of a certain type A and a quotient of a certain type B, *i.e.*, the case of a trivial QPER in which the zig-zag morphism is a surjection from A to B.

Modern approaches to proof transfer rely on univalence, either as an axiom, in the case of univalent parametricity [19] or as a computing primitive [2]. Key ingredients of univalent parametricity were already present in earlier seemingly unpublished work [1], implemented using an ancestor of the MetaCoq library [18].

The columns of Table 1 lists these tools in chronological order, and indicates when the features listed as lines are available (✓), not available (✗) or only partially available (✐). Transfer along *heterogeneous relations*, and while the oldest tool operates via a monolithic translation of an input proof term, others rather prove an *internal* implication lemma. *Anticipation* [19] refers to the need to define a dedicated structure for the signature to be transported. *Binders* (∀) can prevent transfer, as well as *dependent types*, which require univalence.

	[16]	[17]	[7]	[14]	[21]	[19]	[2]	[4]	Trocq
Heterogen. rel.	✓	✗	✓	✓	✓	✓	✓	✓	✓
Internal	✗	✓	✓	✓	✓	✓	✓	✓	✓
No anticipation	✓	✓	✓	✓	✓	✓	✗	✓	✓
Under ∀	✓	✓	✗	✓	✓	✓	✓	✓	✓
Dep. types	✓	✗	✗	✗	✗	✓	✓	✗	✓
Univalence-free	✓	✓	✓	✓	✓	✗	✗	✓	✓
Subrelations	✗	✓	✗	✗	✗	✗	✗	✗	✐
QERs	✗	✐	✐	✐	✐	✗	✓	✗	✐
Subtyping	✗	✗	✐	✐	✐	✗	✗	✐	✐
	Coq	Coq	Coq	Isabelle/HOL	Coq	HoTT	CubicalAgda	Coq	Coq or HoTT

Table 1. Comparison of proof transfer automation devices

Acknowledgments. The authors would like to thank Enrico Tassi and the anonymous reviewers. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101001995).

References

1. Anand, A., Morrisett, G.: Revisiting parametricity: Inductives and uniformity of propositions (2017), `http://arxiv.org/abs/1705.01163`
2. Angiuli, C., Cavallo, E., Mörtberg, A., Zeuner, M.: Internalizing representation independence with univalence. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021)
3. Barthe, G., Pons, O.: Type isomorphisms and proof reuse in dependent type theory. In: FoSSaCS. LNCS, vol. 2030, pp. 57–71. Springer (2001)
4. Blot, V., Cousineau, D., Crance, E., de Prisque, L.D., Keller, C., Mahboubi, A., Vial, P.: Compositional pre-processing for automated reasoning in dependent type theory. In: CPP. pp. 63–77. ACM (2023)
5. Cohen, C., Crance, E., Mahboubi, A.: Trocq: proof transfer for free, with or without univalence. In: Weirich, S. (ed.) Programming Languages and Systems. LNCS, vol. 14576, pp. 239–268. Springer, Cham (2024). `https://doi.org/10.1007/978-3-031-57262-3_10`
6. Cohen, C., Crance, E., Mahboubi, A.: coq-community/trocq: Trocq 0.1.5 (Jan 2024). `https://doi.org/10.5281/zenodo.10563382`
7. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: CPP. LNCS, vol. 8307, pp. 147–162. Springer (2013)
8. Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to computational algebra in coq. In: ITP. LNCS, vol. 7406, pp. 83–98. Springer (2012)
9. Dunchev, C., Guidi, F., Coen, C.S., Tassi, E.: ELPI: fast, embeddable, λProlog interpreter. In: LPAR. LNCS, vol. 9450, pp. 460–468. Springer (2015)
10. Frühwirth, T., Raiser, F.: Constraint handling rules: Compilation, execution, and analysis (2011)
11. Haftmann, F., Krauss, A., Kuncar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: ITP. LNCS, vol. 7998, pp. 100–115. Springer (2013)
12. Huffman, B., Kuncar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: CPP. LNCS, vol. 8307, pp. 131–146. Springer (2013)
13. Krishnaswami, N.R., Dreyer, D.: Internalizing relational parametricity in the extensional calculus of constructions. In: CSL. LIPIcs, vol. 23, pp. 432–451. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013)
14. Lammich, P.: Automatic data refinement. In: ITP. LNCS, vol. 7998, pp. 84–99. Springer (2013)
15. Lammich, P., Lochbihler, A.: Automatic refinement to efficient data structures: A comparison of two approaches. J. Autom. Reason. **63**(1), 53–94 (2019)
16. Magaud, N.: Changing data representation within the Coq System. In: TPHOLs. LNCS, vol. 2758, pp. 87–102. Springer (2003)
17. Sozeau, M.: A new look at generalized rewriting in type theory. J. Formaliz. Reason. **2**(1), 41–62 (2009)
18. Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., Winterhalter, T.: The metacoq project. J. Autom. Reason. **64**(5), 947–999 (2020)
19. Tabareau, N., Tanter, É., Sozeau, M.: The marriage of univalence and parametricity. Journal of the ACM (JACM) **68**(1), 1–44 (2021)
20. Tassi, E.: Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λprolog). In: CoqPL (January 2018), `https://hal.inria.fr/hal-01637063`
21. Zimmermann, T., Herbelin, H.: Automatic and transparent transfer of theorems along isomorphisms in the coq proof assistant. In: CICM (Work in Progress). pp. 50–62 (2015)

Observational Equality Meets CIC

Loïc Pujet[1] and Nicolas Tabareau[2](✉)

[1] University of Stockholm, Stockholm, Sweden
[2] Inria, Paris, France
nicolas.tabareau@inria.fr

Abstract. Equality is at the heart of dependent type theory, as it plays a fundamental role in specifications and mathematical reasoning. The standard way to handle it in mainstream proof assistants such as Agda, Lean or Coq is based on Martin-Löf's identity type, which comes straight out of the '70s—its elegance and simplicity have earned it a long-standing use, despite a major discrepancy with traditional mathematical formulations: it does not satisfy any extensionality principles. Recently, the work on observational equality has regained interest as a new way to encode equality in proof assistants that support a universe of definitionally proof-irrelevant propositions; however it has yet to be integrated in any major proof assistant, because it is not fully compatible with another important feature of type theory: indexed inductive types. In this paper, we propose a systematic integration of indexed inductive types with an observational equality, and show that this integration can only be completely satisfactory if the observational equality satisfies the computational rule of Martin-Löf's identity type. The second contribution of this paper is a formal proof that this additional computation rule, although not present in previous works on observational equality, can be integrated to the system without compromising the decidability of conversion.

1 Introduction

Equality is a fundamental part of mathematical reasoning and formal specification, and it is therefore at the heart of any proof assistant. In Martin-Löf Type Theory [17], it is expressed with the *identity type*, which is characterized by two elegantly simple principles: equality is reflexive, and an equality proof cannot be told apart from a proof by reflexivity from inside the theory (this is known as the *J rule*, or *transport*). From these two principles, it is possible to show that the identity type is symmetric, transitive, and even that it satisfies all the laws of a higher groupoid [9]. This Martin-Löf identity type serves as the base for the interpretation of equality in the proof assistants Agda, Coq and Lean.

Unfortunately, this alluring formulation suffers from serious drawbacks: it is impossible to prove extensionality principles for the identity type, and the uniform definition makes it difficult to integrate types for which the equality relation is specified *ad hoc*, such as quotient types. In practice however, quotient types and extensionality principles are pervasive in mathematics; in particular the principle of function extensionality—which says that two functions are equal

© The Author(s) 2024
S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 275–301, 2024.
https://doi.org/10.1007/978-3-031-57262-3_12

when they are equal at every point—is taken for granted by most mathematicians and computer scientists. While it is possible to postulate those extensionality principles as axioms, this comes at the price of blocking computation for the transport operator.

In order to improve this sorry state of affairs, the most natural solution is to go back at the root of the problem and replace the dysfunctional identity type with a better-behaved alternative, for instance with the *observational equality* of [6]. Unlike Martin-Löf's identity type, the observational equality has a specific definition for each type former, so that the definition of quotient types becomes straightforward and extensionality principles can be added without too much trouble. There is some amount of freedom in the precise implementation of this idea; in this work we will build upon the recently proposed system CC^{obs} [23]. Thus in CC^{obs}, every type A is equipped with an observational equality $t \sim_A u$, defined as a proof-irrelevant proposition with a reflexivity proof written `refl`. The system also provides a primitive type-casting operator `cast` $A\ B\ e\ t$ that can be used to coerce a term t of type A to the type B, given a proof e that these two types are observationally equal. This type-casting operator can then be used to derive the J rule for the observational equality, which ensures that it is a reasonable notion of equality and thus a good candidate for an implementation in a proof assistant.

But even though the idea has been around for almost two decades, none of the mainstream proof assistants supports the observational equality as of 2023. One possible reason is that it is not so easy to integrate it with the sophisticated type systems of modern proof assistants such as Agda, Coq and Lean, and in particular with their system of inductive definitions. Thus, the first contribution of this work is to extend CC^{obs} with the indexed inductive types of Coq and their computation rules, resulting in a system that we call $\mathsf{CIC}^{\text{obs}}$. We do this by exhibiting a general mechanism that distinguishes casts on parameters which can be propagated in the arguments of constructors, and casts on indices which are blocked and create new normal forms. Therefore, the indexed inductive types of $\mathsf{CIC}^{\text{obs}}$ can contain more inhabitants than their counterparts in CIC; they only coincide when indices are taken in a type with decidable equality (*e.g.,* natural numbers in the case of vectors). Additionally, in order to properly handle the propagation of the casts, we give a general account of which equalities can be deduced from an observational equality between two instances $I\ \vec{x}$ and $I\ \vec{y}$ of the same inductive type. The correct rule is slightly more subtle than the injectivity of type formers—in particular, when a parameter of I is not used in the definition of the constructors of the inductive type, the equality of the two instances does not imply the equality of the parameter.

Our treatment of indices is based on *Fordism*, a technique that makes use of the equality type to reduce indexed inductive definitions to parametrized definitions. Its usefulness in an observational context has already been noted in [5], but it should be emphasized that the computational faithfulness of Fordism crucially relies on the computation rule for transport, which is weakened in the system of [23]: the encoding of transport *via* the `cast` operator does not compute on reflexivity proofs as well as the eliminator of Martin-Löf's identity type. More precisely, in $\mathsf{CC}^{\mathrm{obs}}$ it is possible to prove that the propositional equality

$$\texttt{cast}\ A\ A\ (\texttt{refl}\ A)\ t \quad \sim_A \quad t$$

is inhabited for any type A, but the equality does not hold definitionally. This seemingly harmless difference implies that the observational equality of $\mathsf{CC}^{\mathrm{obs}}$ cannot be used to encode the indexed definitions of CIC. This issue is well-known, and previous work [22] introduced an auxiliary equality defined as a quotient type to recover this computation rule at the cost of the definitional uniqueness of identity proofs (UIP), in a way that is reminiscent of Swan's identity type in cubical type theories [25]. In our new system $\mathsf{CIC}^{\mathrm{obs}}$, we go a step further and show that the tension can be fully resolved by using the idea of [4] that under certain conditions, definitional equalities that hold on closed terms can be extended to open terms by adding *new definitional equations on neutral terms.* Indeed, the failure of the computation rule for transport only occurs on open terms, since `cast` computes on types and terms instead of the equality proof. For instance, in the case of the identity cast on natural numbers it is already true in $\mathsf{CC}^{\mathrm{obs}}$ that $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ (\texttt{refl}\ \mathbb{N})\ 32 \equiv 32$, and similarly for any closed natural number—this is a direct consequence of the canonicity theorem proved in [22]. What is missing is the equation $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ (\texttt{refl}\ \mathbb{N})\ n \equiv n$ when n is a neutral term, in particular a variable. Thus the problem to be addressed is:

"*Can we add those new definitional equations while keeping conversion and type checking decidable?*"

In the case of the type of natural numbers, it is very tempting to transform this equation into a new reduction rule $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n \Rightarrow n$. However the case of two neutral types `A` and `B` seems more delicate, since the corresponding rule $\texttt{cast}\ A\ B\ e\ t \Rightarrow t$ should fire only when `A` and `B` are convertible, and reduction rules that rely on a conversion premise are still poorly understood [28,1].

Fortunately, this is not the only way to support the desired definitional equality. Coming back to the case of natural numbers, if n is neutral then neither n nor $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n$ will trigger the reduction of an eliminator; therefore the decision that $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n \equiv n$ can be deferred to equality checking after reduction, in the same way that one usually decides η-equality for functions. The second contribution of this paper is a formal proof that this algorithm does indeed lead to a sound and complete decision procedure for conversion. The argument is formalized in Agda, (see Section 8), following previous work on logical relations [2,22,23].

Related work The first proof assistant to implement an observational equality was the now-defunct Epigram 2 [19]. Although it did not have a primitive scheme for inductive definitions *à la* Coq, Epigram 2 had support for indexed W-types based on a fancy notion of containers, and its equality type did implement the computation rule on reflexivity, meaning that the user could use it to encode indexed definitions using Fordism. The normalization and consistency of Epigram 2 is justified with an inductive-recursive embedding into Agda, but this embedding does not account for the computation rule on reflexivity, which is only conjectured not to break normalization and decidability.

In the world of cubical type theories, more attention has been paid to the definition of general (higher) inductive types [10]. There, the situation is complicated by the fact that transport for the cubical equality does not supports definitional computation on reflexivity as of today (this is known as the *regularity* problem), thus the Fordism encoding cannot be used straightforwardly. Instead, Cavallo and Harper add a *fcoe* constructor to their indexed inductive types in order to keep track of the coercions on indices, and they obtain that an inhabitant of an inductive type in normal form is a chain of *fcoe* applied to a canonical constructor. These inductive definitions have been implemented in Cubical Agda [27] and have been used to develop a sizeable standard library.

2 Observational Equality Meets CIC at Work

The Calculus of Inductive Constructions (CIC), which is the theoretical foundation of the proof assistants Coq and Lean, includes a powerful scheme for inductive definitions [21]. It supports parameters, indices and recursive definitions, but also more exotic features such as mutually defined or nested families. The high level of generality of this scheme allows it to subsume types as diverse as the natural numbers, Σ-types, W-types, and Martin-Löf's identity type. If we are to extend Coq with an observational equality, then we need to understand how it interacts with these inductive definitions, and to devise suitable computation rules. While some of these rules are self-evident, others will turn out to be more delicate. In order to help the reader build their intuition, we study the observational version of three common inductive types: lists, Martin-Löf's identity type and vectors.

2.1 Lists

We start with a brief look at the datatype of lists parametrized by an arbitrary type. Its definition in Coq might look something like this:

```
Inductive list (A:Type) : Type :=
| nil : list A
| cons : A → list A → list A.
```

The basic rules of the CIC already provide us with an eliminator and computation rules for this inductive type. In the language of Coq, these are implemented *via* a pattern-matching construction (`match`) and a guarded fixpoint operator (`fix`) [11]. But in an observational type theory, we need more than just the rules for introduction, elimination and computation—every type former should come with three additional ingredients: a definition of the observational equality between inhabitants, a definition of the observational equality between two instances of the type, and computation rules for `cast`.

There is some leeway for the definition of the observational equality on any given type. In its original version and most of the subsequent literature, the observational equality type itself evaluates to a domain-specific equality, meaning that a proof of equality between two functions is *definitionally* the same as a proof of pointwise equality [6,23]. On the other hand, it is possible to implement an observational type theory in which the equality type does not reduce, but is instead equipped with primitive operators that can be used to convert (for instance) a pointwise equality of functions into an equality [7]. In this paper, we will go with the second approach, as it turns out to be better suited for an implementation in Coq.

Now, what operators should we add in the case of lists? Obviously, two lists should be observationally equal if and only if they are either both empty, or have equal heads and recursively equal tails. But as it turns out, this logical equivalence is already derivable from the induction scheme for lists and the J eliminator for the observational equality—just like we would prove it in plain intensional Martin-Löf Type Theory (MLTT). Therefore, we do not need to characterize the equality between lists any further. This stems from the fact that inductive types are free algebras, and do not need any sort of quotienting in their construction. The observational equality between inhabitants of the universe, on the other hand, does not profit from such an induction principle. Thus we add a new operator to our theory, which takes an equality between two list types and "projects" out an equality between the underlying types:

$$\texttt{eq-list} : \texttt{list A} \sim \texttt{list B} \rightarrow \texttt{A} \sim \texttt{B}.$$

This principle is necessary, because a proof of equality between `list` A and `list` B should allow us to coerce a list of elements of A into a list of elements of B, and thus in particular it should allow us to coerce from A to B. Since this implication is in fact a logical equivalence (the converse direction is provable from the J eliminator), it does indeed fully determine the observational equality between list types. Finally, we need to add rules that explain how `cast` computes on lists. Unlike the computation rules for the observational equality types, these are very much necessary, unless we are fine with having stuck computations in an empty context. Here, there is only one natural choice: casting a constructor of `list A` should evaluate to the corresponding constructor of `list B`.

$$\begin{aligned}
\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ \texttt{nil} &\equiv \texttt{nil}\\
\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ (\texttt{cons}\,a\,l) &\equiv \texttt{cons}\,(\texttt{cast}\ A\ B\ (\texttt{eq-list}\ e)\ a)\\
&\qquad\quad (\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ l)
\end{aligned}$$

Remark that in the case of a non-empty list, we need the `eq-list` axiom in order to apply `cast` to the head of the list. *Voilà*, this is all it takes for an observational type theory with lists. With this example under our belt, we now move on to a more sophisticated example.

2.2 Indices and Fordism

The next layer of complexity offered by the scheme of Coq is indices. Here, the story gets more complicated, as indexed definitions gain new inhabitants in the presence of the observational equality. To see this, consider Martin-Löf's identity type, which is the prototypical example of an indexed inductive definition:

```
Inductive Id (A : Type) (x : A) : A → Type := id_refl : Id A x x.
```

In intensional type theory, it is well-known that this equality type does not satisfy the principle of function extensionality. But in our observational type theory, it turns out we can to prove that Martin-Löf's identity type is *logically equivalent* to the observational equality (we can use the `cast` operator in one direction, and the induction principle for `Id` in the other direction). In particular, the principle of function extensionality is now provable for `Id`! As convenient as it might sound, it also implies that we can get an inhabitant of the type `Id` $(\mathbb{N} \to \mathbb{N})$ $(\lambda n.1 + n)$ $(\lambda n.n + 1)$ in the empty context, since the two functions are extensionally equal. But this inhabitant cannot be definitionally equal to `id_refl`, as the two functions are not convertible. From this, we deduce that the closed inhabitants of an indexed inductive type may include more than the *canonical* ones, *i.e.,* those that can be built out of the constructors of the inductive type.

In order to get a better grasp on these noncanonical inhabitants, we can turn our attention to *Fordism*. This technique was invented by Coquand for his work on the proof assistant half in the 1990s, as a way to reduce indexed inductive types to parametrized inductive types and an equality type. The name Fordism first appeared in [18], in reference to a famous quote by Henry Ford: "A customer can have a car painted any color he wants as long as it's black". Let us look at the construction at work on the inductive definition of vectors, which is a little less barebones than the inductive identity type:

```
Inductive vec (A:Type) : ℕ → Type :=
| vnil : vec A 0
| vcons : ∀ m, A → vec A m → vec A (S m).
```

Vectors are basically lists with an additional index that makes their length available in the type, ensuring that a vector of type `vec A n` contains `n` elements. In order to get the *forded* version of vectors, we modify their definition so that the index becomes a parameter, and the two constructors gain a new argument:

```
Inductive vec𝔽 (A:Type) (n : ℕ) : Type :=
| vnil𝔽 : n ∼ℕ 0 → vec𝔽 A n
| vcons𝔽 : ∀ m, A → vec𝔽 A m → n ∼ℕ S m → vec𝔽 A n.
```

Remark that a forded empty vector `vnil`$_\mathbb{F}$ `e` can have *a priori* the type `vec A n` for any n, except that `e` is a witness that `n` is equal to 0. An empty vector can have any size you want, as long as it's zero! The point of Fordism is that the induction principle of `vec` can be derived for `vec`$_\mathbb{F}$, by combining the induction principle provided by the CIC for `vec`$_\mathbb{F}$ and the eliminator of the equality:

```
vec_elim (A : Type) (P : ∀ n : ℕ, vec𝔽 A n → Type) :
   P 0 (vnil𝔽 0 refl) →
   (∀ (m : ℕ) (a : A) (v : vec𝔽 A m), P m v → P (S m) (vcons𝔽 (S m) m a v refl)) →
   ∀ (n : ℕ) (v : vec𝔽 A n), P n v.
vec_elim A P Pnil Pcons n (vnil𝔽 n e) ≡
  cast (P 0 (vnil𝔽 0 refl)) (P n (vnil𝔽 n e)) (vnil_ap A e) Pnil.
vec_elim A P Pnil Pcons n (vcons𝔽 n m a v e) ≡
  cast (P (S m) (vcons𝔽 (S m) m a v refl)) (P n (vcons𝔽 n m a v e))
```

```
(vcons_ap A m a e v) (Pcons m a v (vec_elim A P Pnil Pcons m v)).
```

Here, we used implicit arguments for `refl` and we used two auxiliary definitions $\texttt{vnil}_{ap}$ and $\texttt{vcons}_{ap}$ showing that functions preserve equalities. Furthermore, *if the* `cast` *operator satisfies the computation rule on reflexivity*, then the induction principle provided by the Fordism transformation satisfies the same computation rules as the standard induction principle for indexed inductive types. Thus, Fordism can serve as a recipe for the implementation of indexed inductive types, as long as we know how to handle parametrized inductive types and have an equality that computes on reflexivity.

Additionally, this transformation sheds some light on the noncanonical elements of indexed inductive types: in CIC, the only closed proof of equality is a proof by reflexivity, thus the inhabitants of $\texttt{vec}_{\mathbb{F}}$ `A n` in the empty context behave exactly like the canonical inhabitants of `vec A n`. But in an observational type theory, there are many proofs of equality in the empty context (think for example of a proof of equality between two functions that are not convertible, but extensionally equal) which give rise to new elements. These elements can be obtained by casting a canonical inhabitant to a type with a different (but observationally equal) index, and they cannot be eliminated away in general.[3]

2.3 Parameters and Equalities

Now that we know how to handle indexed types, we can revisit Martin-Löf's identity type, which plays an important role in CIC. After the Fordism transformation, its definition looks like this:

$$\texttt{Inductive Id}_{\mathbb{F}}\ (\texttt{A} : \texttt{Type})\ (\texttt{x y} : \texttt{A}) : \texttt{Type} := \texttt{id\_refl}_{\mathbb{F}} : \texttt{x} \sim_A \texttt{y} \to \texttt{Id}_{\mathbb{F}}\ \texttt{A x y}.$$

As we want to incorporate this type into our observational theory, we apply the standard recipe: we need a definition of the observational equality between inhabitants of $\texttt{Id}_{\mathbb{F}}$, a definition of the observational equality between two instances of $\texttt{Id}_{\mathbb{F}}$, and computation rules for the `cast` operator. The first one is easy, as we can prove that any two inhabitants of $\texttt{Id}_{\mathbb{F}}$ `A x y` are equal: by induction, we only need to prove it for elements of the form $\texttt{id\_refl}_{\mathbb{F}}$ `e`, with `e` being a proof of $\texttt{x} \sim_A \texttt{y}$. But the observational equality is definitionally proof-irrelevant, so this is true by reflexivity. In other words, the principle of *uniqueness of identity proofs* (UIP) is provable for the inductive identity type in observational type theory, in stark contrast with MLTT or CIC. Thus, we do not need any further characterization of the observational equality between inhabitants of $\texttt{Id}_{\mathbb{F}}$.

On the other hand, the definition of the observational equality between two instances of the identity type $\texttt{Id}_{\mathbb{F}}$ `A x y` and $\texttt{Id}_{\mathbb{F}}$ `A' x' y'` makes for another interesting story. From our study of lists, it might be tempting to extrapolate that an observational equality between two instances of a parametrized inductive

[3] In the case of vectors, it is possible to find alternative encodings that do not have these new canonical elements, because the equality between indices is decidable in the empty context. However, we aim at a systematic and uniform treatment of indexed inductive types, so we will not consider this option.

datatype should imply an equality between the parameters, or in the special case of $\mathtt{Id}_{\mathbb{F}}$, that we get the following principle:

$$\mathtt{Id}_{\mathbb{F}}\ \mathtt{A\ x\ y} \sim \mathtt{Id}_{\mathbb{F}}\ \mathtt{B\ z\ w} \to \exists\ (\mathtt{e} : \mathtt{A} \sim \mathtt{B}), (\mathtt{cast\ A\ B\ e\ x} \sim \mathtt{z}) \wedge (\mathtt{cast\ A\ B\ e\ y} \sim \mathtt{w})$$

This means that parametrized inductive definitions are *injective* functions from the type of parameters to the universe. Unfortunately, this idea turns out to be incompatible with the rules of CIC. Indeed, according to these rules the inductive equality `Id A x y` should live in the lowest universe, since it has only one constructor with no arguments. But then if `A` is a large type, we get an injective function from `A` into the lowest universe, which is potentially inconsistent—for instance, consider the following function:

$$\mathtt{inj}\ (\mathtt{X} : \mathtt{Type} \to \mathtt{Type}) := \mathtt{Id}_{\mathbb{F}}\ (\mathtt{Type} \to \mathtt{Type})\ \mathtt{X\ X}$$

If the $\mathtt{Id}_{\mathbb{F}}$ type former is injective, then `inj` is an injection of `Type` → `Type` into `Type`, from which we can encode Russell's paradox and derive an inconsistency for CIC [20]. Thus, if we really want to have this injectivity of parameters, we need to modify the rules of our theory so that inductive definitions are only allowed in a universe that is sufficiently large to accommodate their parameters. But this is not exactly reasonable: this would mean that we cannot abstract over the definition of an inductive type using Coq's sections mechanism, since section variables are translated to inductive parameters. In other words, inductive definitions would only make sense in closed contexts.

In order to avoid such a serious drawback, we will use a completely different characterization for the observational equality between inductive types. After all, what do we need these axioms for? The answer is simple: we need some observational equalities to put in the computation rules for the `cast` operator.

$$\mathtt{cast}\ (\mathtt{Id}_{\mathbb{F}}\ \mathtt{A\ x\ y})\ (\mathtt{Id}_{\mathbb{F}}\ \mathtt{B\ z\ w})\ \mathtt{e}\ (\mathtt{id\_refl}_{\mathbb{F}}\ \mathtt{e'}) \equiv \ldots$$

For inductive types, these computation rules are very systematic: when `cast` is applied to a constructor, then it should naturally reduce to the corresponding constructor of the target inductive. Thus, we need to produce an inhabitant of $\mathtt{x'} \sim_{A'} \mathtt{y'}$ from an inhabitant of $\mathtt{x} \sim_{A} \mathtt{y}$. This is a job for the `cast` operator:

$$\mathtt{cast}\ (\mathtt{Id}_{\mathbb{F}}\ \mathtt{A\ x\ y})\ (\mathtt{Id}_{\mathbb{F}}\ \mathtt{B\ z\ w})\ \mathtt{e}\ (\mathtt{id\_refl}_{\mathbb{F}}\ \mathtt{h}) \equiv \mathtt{id\_refl}_{\mathbb{F}}\ (\mathtt{cast}\ (\mathtt{x} \sim \mathtt{y})\ (\mathtt{z} \sim \mathtt{w})\ ?\ \mathtt{h}).$$

In order to fill the question mark hole, we need a proof of observational equality between the two observational equality types. Since all we have is a proof of equality between $\mathtt{Id}_{\mathbb{F}}\ \mathtt{A\ x\ y}$ and $\mathtt{Id}_{\mathbb{F}}\ \mathtt{B\ z\ w}$, we need something to extract the desired equality. The injectivity of the inductive types is *sufficient* for this purpose, but it is not *necessary*. Instead, we can go for the bare minimum: an observational equality between two instances of the same inductive definition should imply the equality of all their argument contexts, and nothing more. In the case of the inductive $\mathtt{Id}_{\mathbb{F}}$, it means that we get the following projection:

$$\mathtt{eq{-}Id}_{\mathbb{F}} : \mathtt{Id}_{\mathbb{F}}\ \mathtt{A\ x\ y} \sim \mathtt{Id}_{\mathbb{F}}\ \mathtt{B\ z\ w} \to\ (\mathtt{x} \sim \mathtt{y}) \sim (\mathtt{z} \sim \mathtt{w}).$$

As we will prove in Section 6, this is enough to get an identity type that lives in the lowest universe without endangering the consistency of the theory.

i, j	$\in$	$\mathbb{N}$	Universe levels
s	$::=$	$\mathcal{U}_i \mid \Omega$	Universes
Γ, Δ	$::=$	$\bullet \mid \Gamma, x : A : s$	Contexts
t, u, m, n, e, A, B	$::=$	$x \mid s$	Variables and Universes
	$\mid$	$\lambda(x : A).\, t \mid t\ u \mid \Pi^{s,s'}(x : A).\, B$	Dependent products
	$\mid$	$\bot\texttt{-elim}\ A\ t \mid \bot$	Empty type
	$\mid$	$t \sim_A u \mid \texttt{refl}\ t \mid \texttt{transport}\ A\ t\ B\ u\ t'\ e$	Observational equality
	$\mid$	$\texttt{cast}\ A\ B\ e\ t$	Type cast
	$\mid$	$\Pi^1_\epsilon \mid \Pi^2_\epsilon \mid \Omega_{\text{ext}} \mid \Pi_{\text{ext}}$	Properties of Equality

Fig. 1: Syntax for the negative fragment of $\mathsf{CIC}^{\text{obs}}$ *[Untyped.agda]*

3 CIC$^{\text{obs}}$ with Martin-Löf's Computation Rule

At this stage, we have a clear roadmap for our observational type theory with inductive types: first, we need a system with a `cast` operator that computes on proofs by reflexivity. Then, we handle parametrized inductive types with projection functions for the equality types and computation rules for `cast`, and finally, we can take care of indexed inductive types with some syntactic sugar around the Fordism transformation.

We are now in position to define $\mathsf{CIC}^{\text{obs}}$, the observational type theory that will serve as our theoretical framework. It is based on the system CC^{obs} of [23], but with a few tweaks; the most important one being the additional computation rule for the `cast` operator on reflexivity proofs. In this section, we give a brief presentation of the syntax, typing rules and declarative conversion for the core of the type theory, with an emphasis on the points that differ from CC^{obs}, before defining the scheme for inductive definitions in Section 5. All the definitions in the figures follows closely our Agda formalization. We refer to files in the formalization as *[myFile.agda]*.

3.1 The Syntax of CIC$^{\text{obs}}$

The syntax of the sorts, contexts, terms and types of $\mathsf{CIC}^{\text{obs}}$ is specified in Fig. 1. The sorts of our system are divided into a predicative hierarchy $(\mathcal{U}_i)_{i\in\mathbb{N}}$ which mirrors the `Type` hierarchy of Coq, and an impredicative sort Ω of proof-irrelevant propositions which corresponds to Coq's `SProp`. The base types are the false proposition $\bot$, the observational equality $t \sim_A u$ and the dependent function type $\Pi^{s,s'}(x : A).\, B$. For the sake of readability, we will frequently drop the sort annotations on dependent products when they can be inferred from the context, and when B does not depend on A, we write $A \to B$ instead of $\Pi(x : A).\, B$. In addition to these basic types, our theory also includes a definition scheme for indexed inductive types, that can be used to extend the syntax with new types and terms (cf. Section 5).

Compared to the system CC^{obs} of [23], we add four new primitives Π^1_ϵ , Π^2_ϵ , Ω_{ext} and Π_{ext}, whose role is to provide the properties of the observational

Eq-Ω
$$\frac{\Gamma \vdash A : \Omega \qquad \Gamma \vdash B : \Omega}{\Gamma \vdash \Omega_{\mathrm{ext}} : (A \to B) \to (B \to A) \to A \sim_{\Omega} B}$$

Eq-Fun
$$\frac{\Gamma \vdash A : s \quad \Gamma, x : A : s \vdash B : \mathcal{U}_i \quad \Gamma \vdash f, g : \Pi(x : A).\, B : \mathcal{R}(\mathcal{U}_i, s)}{\Gamma \vdash \Pi_{\mathrm{ext}} : \Pi(x : A).\, f\ x \sim_B g\ x \to f \sim_{\Pi A B} g}$$

Eq-Π$_1$
$$\frac{\Gamma \vdash A, A' : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A' \vdash B' : s'}{\Gamma \vdash \Pi^1_\epsilon : \Pi(x : A).\, B \sim_{\mathcal{R}(s,s')} \Pi(x : A').\, B' \to A' \sim_s A}$$

Eq-Π$_2$
$$\frac{\Gamma \vdash A, A' : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A' \vdash B' : s'}{\Gamma \vdash \Pi^2_\epsilon : \Pi(e : \_).\, \Pi(a' : A').\, B[x := \mathtt{cast}\ A'\ A\ (\Pi^1_\epsilon\ e)\ a'] \sim_{s'} B'[x := a']}$$

Fig. 2: $\mathsf{CIC}^{\mathrm{obs}}$ rules for characterizing the observational equality *[Typed.agda]*

equality which were previously given as computation rules. For instance, in the system of [23] an equality between two function types evaluates to a Σ-type that contains equalities of the domain and codomain, while in our new system these two equalities are obtained by applying Π^1_ϵ and Π^2_ϵ to the proof of equality between function types. Replacing computations with these new primitives does *not* endanger the computational properties of our theory, since they only ever produce computationally irrelevant equality proofs. Plus, it results in a more elegant system that does not need a primitive Σ-type; this way of handling the properties of the observational equality will be especially convenient when dealing with inductive definitions, where equalities between types imply complex telescopes of equalities which would be cumbersome to express with nested Σ-types.

3.2 The Typing Rules of $\mathsf{CIC}^{\mathrm{obs}}$

The typing rules of $\mathsf{CIC}^{\mathrm{obs}}$ are based on five judgments:

$\vdash \Gamma$	Γ is a well-formed context,
$\Gamma \vdash A : s$	A is a well-formed type of sort s in Γ,
$\Gamma \vdash t : A : s$	t is a term of type A in sort s in Γ,
$\Gamma \vdash A \equiv B : s$	A and B are convertible types of sort s in Γ, and
$\Gamma \vdash t \equiv u : A : s$	t and u are convertible terms of type A in Γ.

In all the judgments, s denotes either $\mathcal{U}_i$ or Ω. Note that since every universe has a type, the well-formedness judgments for types $\Gamma \vdash A : s$ (and convertibility judgments of types) can be seen as special cases of typing judgments for terms $\Gamma \vdash A : s : s'$ for a suitable s', but we keep the type-level judgments to avoid writing unnecessarily many sort variables.

The rules for universes, dependent function types, and the empty type are taken directly from [23], so we only give a brief overview here. The complete set of rules is available in *[Typed.agda]*. We use PTS-style notations [8] to factorize the impredicative and predicative rules for universes and dependent products: the formation rule for universes states that both $\mathcal{U}_i$ and Ω are inhabitants of a higher universe, as described by the relations

$$\mathcal{A}(\mathcal{U}_i,\mathcal{U}_j) := j = i+1 \qquad \mathcal{A}(\Omega,\mathcal{U}_i) := i = 0.$$

We allow the formation of dependent products with a domain and a codomain that have different sorts. If the codomain is a proof-relevant type, then the dependent product should have a universe level that is the maximum between the level of the domain and that of the codomain. On the other hand, if the codomain is a proposition then the result is a proposition regardless of the size of the domain. This is made formal by using the function $\mathcal{R}(\_,\_)$ defined as

$$\mathcal{R}(s,\Omega) := \Omega \qquad \mathcal{R}(\Omega,\mathcal{U}_i) := \mathcal{U}_i \qquad \mathcal{R}(\mathcal{U}_i,\mathcal{U}_j) := \mathcal{U}_{max(i,j)}.$$

Equality and Type Casts Every proof-relevant type comes equipped with a propositional binary relation, noted $t \sim_A u$ and called the *observational equality*. This type has one introduction rule that turns it into a reflexive relation. Of course, proof-irrelevant types have no use for an observational equality, since any two inhabitants would always be in relation by reflexivity. The observational equality is equipped with two elimination principles, which are called `transp` and `cast`. The former is similar to the J eliminator from MLTT, except that it is restricted to propositional predicates. Elimination into the proof-relevant layer is thus handled by the `cast` operator, which provides coercions between two observationally equal types. It might seem less general than the standard J eliminator, but since equality proofs are definitionally irrelevant, it turns out that a J eliminator for proof-relevant predicates can be derived from the `cast` operator.

As we already mentioned, the extensional properties of the observational equality are given by the primitives Π^1_ϵ , Π^2_ϵ , Ω_{ext} and Π_{ext}: rules Eq-Π_1 and Eq-Π_2 allow us to deduce the equality of domains and codomains from an equality between two dependent functions types, rule Eq-Ω provides propositional extensionality, and rule Eq-Fun provides function extensionality.

3.3 Conversion

The conversion, also called *definitional* equality, is a judgment that relates the terms that are interchangeable in typing derivations. The rules that define the conversion judgment are reproduced in Fig. 3. By definition, conversion is a reflexive, symmetric and transitive relation. It is also closed under congruence (e.g. if $A \equiv A'$ and $B \equiv B'$ then $\Pi(x : A).B \equiv \Pi(x : A').B'$), although we did not reproduce all the corresponding rules in Fig. 3 for the sake of brevity. The conversion judgment is itself subject to the conversion rule (rule Conv-Conv).

As usual, the conversion relation contains the β-equality for proof-relevant applications (rule β-conv), and the η-equality of functions[4] (rule η-Eq). The

[4] The propositional η-equality is actually provable in observational type theory, since it is a special case of the extensionality of functions. Nevertheless, it is still convenient to have as a conversion rule, to get a more flexible system.

$$\textsc{Refl}\ \frac{\Gamma \vdash t : A : \mathcal{U}_i}{\Gamma \vdash t \equiv t : A : \mathcal{U}_i} \qquad \textsc{Sym}\ \frac{\Gamma \vdash t \equiv u : A : \mathcal{U}_i}{\Gamma \vdash u \equiv t : A : \mathcal{U}_i} \qquad \textsc{Trans}\ \frac{\Gamma \vdash t \equiv t' : A : \mathcal{U}_i \qquad \Gamma \vdash t' \equiv u : A : \mathcal{U}_i}{\Gamma \vdash t \equiv u : A : \mathcal{U}_i}$$

$$\eta\text{-}\textsc{Eq}\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash t, u : \Pi^{s,\mathcal{U}_i}(x : A).\, B : \mathcal{R}(s,\mathcal{U}_i) \qquad \Gamma, x : A : s \vdash t\ x \equiv u\ x : B : \mathcal{U}_i}{\Gamma \vdash t \equiv u : \Pi^{s,\mathcal{U}_i}(x : A).\, B : \mathcal{R}(s,\mathcal{U}_i)}$$

$$\textsc{Proof-Irr}\ \frac{\Gamma \vdash t : A : \Omega \qquad \Gamma \vdash u : A : \Omega}{\Gamma \vdash t \equiv u : A : \Omega} \qquad \textsc{Conv-Conv}\ \frac{\Gamma \vdash t \equiv u : A : \mathcal{U}_i \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash t \equiv u : B : \mathcal{U}_i}$$

$$\beta\text{-}\textsc{conv}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A : s \vdash B : \mathcal{U}_i \qquad \Gamma, x : A \vdash t : B : \mathcal{U}_i \qquad \Gamma \vdash u : A : s}{\Gamma \vdash (\lambda(x : A).\, t)\ u \equiv t[x := u] : B[x := u] : \mathcal{U}_i}$$

$$\textsc{Cast-}\Pi\ \frac{\begin{array}{c}\Gamma \vdash A : s \qquad \Gamma \vdash A' : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A' \vdash B' : s' \\ \Gamma \vdash e : \Pi(x : A).\, B \sim \Pi(x : A').\, B' : \Omega \qquad \Gamma \vdash f : \Pi(x : A).\, B \qquad a := \mathtt{cast}\ A'\ A\ (\Pi^1_\epsilon\ e)\ a'\end{array}}{\Gamma \vdash \begin{array}{l}\mathtt{cast}\ (\Pi(x : A).\, B)\ (\Pi(x : A').\, B')\ e\ f \equiv \\ \lambda(a' : A').\, \mathtt{cast}\ (B[x := a])\ (B'[x := a'])\ (\Pi^2_\epsilon\ e\ a')\ (f\ a)\end{array} : \Pi(x : A').\, B' : \mathcal{R}(s,s')}$$

$$\textsc{Cast-Refl}\ \frac{\Gamma \vdash A \equiv B : s \qquad \Gamma \vdash e : A \sim_s B \qquad \Gamma \vdash t : A : s}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \equiv t : B : s}$$

Fig. 3: $\mathsf{CIC}^{\text{obs}}$ Conversion Rules (except congruence rules) *[Typed.agda]*

rule Proof-Irr reflects the computational irrelevance of the propositions: any two inhabitants of the same proposition are deemed convertible. Additionally, the conversion relation also includes the computation rules for the pattern-matching of inductive constructors that we will define in Section 5.

Then, we have the rules describing the behaviour of the `cast` operator on each type. The rule Cast-Π is standard; it says that a cast function evaluates to a function that casts its argument, applies the original function, and then casts back the result. Note that this rule needs the two projections Π^1_ϵ and Π^2_ϵ to get equality between the domains and the co-domains. Likewise, every declaration of an inductive type will add a handful of computation rules for the `cast` operator. Last but not least, the rule Cast-Refl is the main innovation of $\mathsf{CIC}^{\text{obs}}$. It states that `cast` between convertible types can be simplified away, regardless of the proof of equality. This rule plays an important role in ensuring compatibility with the CIC: recall that `cast` can be used to derive a J eliminator for the observational equality—then rule Cast-Refl implies that this eliminator computes on reflexivity proofs, just like the usual eliminator of Martin-Löf's inductive equality.

4 Decidability of Conversion

In this section, we show that conversion is decidable in presence of the rule Cast-Refl for a simplified version of $\mathsf{CIC}^{\text{obs}}$ in which the induction scheme is reduced to the type of natural numbers. Generally speaking, the main source of difficulty for the decidability of conversion in dependent type theory is the transitivity rule—because of it, we have no guarantee that comparing two terms structurally is a complete strategy, since transitivity may be used with an arbitrary intermediate term at any point. If we want a decision procedure, we need to replace this transitivity rule with something more algorithmic.

Our aim is thus to define an equivalent presentation of the conversion for which transitivity is an admissible rule, but is not primitive. This is traditionally achieved by separating the conversion into a notion of weak-head reduction (Section 4.1) and a notion of conversion on neutral terms and weak-head normal forms (Section 4.2). In standard CIC, this strategy is sufficient to get *canonical* derivations of conversion, for which we have a decision procedure: we check the existence of a canonical derivation by first reducing terms to their weak-head normal form, and then comparing their head constructors and making recursive calls on their arguments. The point of this algorithmic definition of conversion is to replace the arbitrary transitivity rules with deterministic computations of weak-head normal forms. Then we can show that transitivity is admissible for conversion on neutral terms and weak-head normal forms. Naturally, this definition requires a proof of normalization of well-typed terms.

In the case of $\mathsf{CIC}^{\mathrm{obs}}$ however, the decision procedure for conversion of neutral terms and weak-head normal forms cannot be defined as a straightforward structural comparison. When the two terms start with `cast`, there are three rules that may apply: either congruence of `cast`, rule Cast-Refl on the left-hand side, or rule Cast-Refl on the right-hand side. This means that the decision procedure (Section 4.3) will have to do some backtracking to explore all possible combinations of congruence of `cast` and Rule Cast-Refl. Fortunately, the search space is bounded as every recursive call is done on a smaller argument.

Finally, to conclude on the decidability of conversion, we need to show that the declarative conversion is equivalent to our algorithmic conversion. For that, we use the logical relation setting of [2] to guarantee that every term can be put in weak-head normal form and that algorithmic conversion is complete with respect to conversion.

Note that our formalized version of $\mathsf{CIC}^{\mathrm{obs}}$ only supports the inductive type of natural numbers, and *not* the full scheme from Section 5. This is due to the setting of the formal proof, which requires the added inductive types to be explicit because Agda's check that the logical relation is well-defined makes use of the strict positivity criterion, which is syntactic and cannot be abstracted away for a generic definition. Nevertheless, we expect that our formal proof can be extended to specific inductive types such as lists or Martin-Löf's identity type, with methods similar to the ones from [3].

4.1 Reduction to Weak-Head Normal Forms

A notion that plays a central role in our normalization procedure is that of a *weak-head normal form* (whnf), which corresponds to a relevant term that cannot be reduced further (Fig. 4). Weak-head normal forms are either terms with a constructor in head position, or *neutral terms* stuck on a variable or

$$\begin{array}{lll}
\text{whnf} & w ::= & N \mid \Pi(x:A).\,B \mid s \mid \mathbb{N} \mid \bot \mid t \sim_A u \mid \lambda(x:A).\,t \mid \mathtt{0} \mid \mathtt{S}\ n \\
\text{neutral} & N ::= & x \mid N\ t \mid \mathtt{\bot{-}elim}\ A\ e \mid \mathtt{\mathbb{N}{-}elim}\ P\ t\ u\ N \\
 & & \mid \mathtt{cast}\ N\ B\ e\ t \mid \mathtt{cast}\ \mathbb{N}\ N\ e\ t \mid \mathtt{cast}\ \Pi^{s,s'}(x:A).\,B\ N\ e\ t \\
 & & \mid \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ N \mid \mathtt{cast}\ w\ w'\ e\ t \\
 & & \quad (\text{where } w, w' \in \{\mathbb{N}, \Pi^{s,s'}(x:A).\,B, s\},\ \mathrm{hd}\, w \neq \mathrm{hd}\, w')
\end{array}$$

Fig. 4: Weak-head normal and neutral forms *[Untyped.agda]*

an elimination of a proof of $\bot$. In other words, neutral terms are weak-head normal forms that should not exist in an empty context. In $\mathsf{CIC}^{\text{obs}}$, inhabitants of a proof-irrelevant type are never considered as whnf, as there is no notion of reduction of proof-irrelevant terms.

This notion of neutral terms is standard, but we need to pay a particular attention to neutral terms for `cast`. They correspond to all forms of cast for which there is no attached reduction rule. Because we assume that `cast` first evaluates its left type argument, then the second and finally its term argument, neutral terms of cast occur either when the first type is neutral, or when the first type is a type constructor and second type is neutral, or when the two types are the same type constructor, but the argument is neutral. Note that the reduction rule for casting a function always fires, so there is no associated neutral term in that case. Finally, casts between two different type constructors are always considered as stuck terms and should be seen as variant of $\mathtt{\bot{-}elim}\ A\ e$ because they correspond to casts based on an inconsistent proof of equality, thus similar to elimination of a proof of $\bot$.

At the heart of the decision procedure for conversion, there is a notion of typed reduction, noted $\Gamma \vdash t \Rightarrow u : A$. Intuitively, reduction corresponds to an orientation of the conversion rule in order to provide a rewrite system for which we can compute normal forms. However, not every conversion corresponds to a reduction rule: turning Rule Cast-Refl into a reduction rule would spawn several critical pairs, and even more annoyingly, its convertibility premise would force us to define reduction mutually with conversion checking. As are not aware of any framework that properly handles this type of circularity, we will sidestep the issue by deferring Cast-Refl to conversion checking, where we only have to deal with neutral terms and weak-head normal forms.

Actually, the purpose of reduction is to compute weak-head normal forms so that conversion rules that are not part of the reduction have only to be checked on weak-head normal forms. We do not detail the standard rules for CIC and focus on the one for `cast` (Fig. 5). The congruence rule for `cast` corresponds to several reduction rules, because we need to be careful to reduce one argument after the other in order, so that weak-head reduction remains deterministic. The reduction rules Cast-Zero, Cast-Suc and Cast-Univ correspond to the rule Cast-Refl where the arguments are instantiated by weak-head normal forms that are not neutral. Indeed, in that case `cast` must reduce. Conversion for `cast` when one of the scrutinees is neutral is deferred to algorithmic conversion.

$$\textsc{Cast-}\Pi\textsc{-red}\quad \frac{\begin{array}{c}\Gamma \vdash A, A' : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A' \vdash B' : s' \\ \Gamma \vdash e : \Pi(x : A).\, B \sim \Pi(x : A').\, B' : \Omega \qquad \Gamma \vdash f : \Pi(x : A).\, B \qquad a := \mathtt{cast}\ A'\ A\ (\Pi^1_\epsilon\ e)\ a'\end{array}}{\Gamma \vdash \begin{array}{l}\mathtt{cast}\ (\Pi(x : A).\, B)\ (\Pi(x : A').\, B')\ e\ f \Rightarrow \\ \lambda(a' : A').\, \mathtt{cast}\ B[x := a]\ B'[x := a']\ (\Pi^2_\epsilon\ e\ a')\ f\ a\end{array} : \Pi(x : A').\, B'}$$

$$\textsc{Cast-Zero}\quad \frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathcal{U}_0} \mathbb{N} : \Omega}{\Gamma \vdash \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ \mathtt{0} \Rightarrow \mathtt{0} : \mathbb{N}} \qquad \textsc{Cast-Suc}\quad \frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathcal{U}_0} \mathbb{N} : \Omega \qquad \Gamma \vdash n : \mathbb{N} : \mathcal{U}_0}{\Gamma \vdash \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ (\mathtt{S}\ n) \Rightarrow \mathtt{S}\ (\mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n) : \mathbb{N}}$$

$$\textsc{Cast-Univ}\quad \frac{\Gamma \vdash e : s \sim_{s'} s \qquad \Gamma \vdash A : s \qquad \mathcal{A}(s, s')}{\Gamma \vdash \mathtt{cast}\ s\ s\ e\ A \Rightarrow A : s} \qquad \textsc{Conv-Red}\quad \frac{\Gamma \vdash t \Rightarrow u : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash t \Rightarrow u : B}$$

$$\textsc{Cast-subst}\quad \frac{\Gamma \vdash A \Rightarrow A' : s \qquad \Gamma \vdash B : s \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t : A : s}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \Rightarrow \mathtt{cast}\ A'\ B\ e\ t : B}$$

$$\textsc{Cast-subst-nf}\quad \frac{\Gamma \vdash A : s \qquad \text{whnf}\ A \qquad \Gamma \vdash B \Rightarrow B' : s \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t : A : s}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \Rightarrow \mathtt{cast}\ A\ B'\ e\ t : B}$$

$$\textsc{Cast-subst-nf-nf}\quad \frac{\Gamma \vdash A, B : s \qquad \text{whnf}\ A \qquad \text{whnf}\ B \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t \Rightarrow u : A}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \Rightarrow \mathtt{cast}\ A\ B\ e\ u : B}$$

Fig. 5: $\mathsf{CIC}^{\text{obs}}$ Reduction Rules (rules for `cast`) *[Typed.agda]*

Note that because reduction is typed, we need to be able to change the type to any convertible one (Rule Conv-Red). Finally, we consider the reflexive transitive closure of reduction, noted $\Gamma \vdash t \Rightarrow^* u : A$.

4.2 Algorithmic Conversion

Algorithmic conversion (Fig. 6) is defined by comparing weak-normal forms and interleaving it with reduction. This way, an algorithmic conversion derivation can be seen as a canonical derivation of declarative conversion, where "transitive cuts" have been eliminated. It is called algorithmic, because it becomes directed by the shape of the terms, and the premises of each rule are on smaller terms. In CIC, it is even the case that at most one rule can be applied, so decidability of algorithmic conversion is pretty direct. In $\mathsf{CIC}^{\text{obs}}$ however, decidability of algorithmic conversion is less direct because there are three rules that can be applied when the head is `cast` on both side. We come back to this difficulty in Section 4.3.

The judgment $\Gamma \vdash t \cong_{ne} u : A$ corresponds to a canonical conversion derivation between two neutral terms t and u at an arbitrary type A while the judgment $\Gamma \vdash t \cong u : A$ corresponds to a canonical derivation of conversion for terms in whnf when the type is also in whnf. This can be understood from a bidirectional perspective because comparison of neutral terms infers an arbitrary type, whereas for other weak-head normal forms, the inferred type is in weak-head normal form. Bidirectional typing [15,16] is traditionally used in type theory to

$$\frac{\Gamma \vdash t, u : A : \Omega}{\Gamma \vdash t \cong_{ne} u : A}\ \textsc{Proof-irr} \qquad \frac{\Gamma \vdash x : A : \mathcal{U}_i}{\Gamma \vdash x \cong_{ne} x : A}\ \textsc{Var-refl}$$

$$\frac{\Gamma \vdash t \cong^{\downarrow}_{ne} u : \Pi^{s,\mathcal{U}_i}(x : A).\, B \qquad \Gamma \vdash a \cong^{\downarrow} b : A}{\Gamma \vdash t\ a \cong_{ne} u\ b : B[x := a]}\ \textsc{app-cong}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \cong A' : s \qquad \Gamma \vdash B' \cong B : s \qquad \Gamma \vdash t \cong^{\downarrow} t' : A \qquad \Gamma \vdash e : A \sim_s B : \Omega \\ \Gamma \vdash e' : A' \sim_s B' : \Omega \qquad \text{neutral}\ (\texttt{cast}\ A\ B\ e\ t) \qquad \text{neutral}\ (\texttt{cast}\ A'\ B'\ e'\ t')\end{array}}{\Gamma \vdash \texttt{cast}\ A\ B\ e\ t \cong_{ne} \texttt{cast}\ A'\ B'\ e'\ t' : B}\ \textsc{Cast-cong}$$

$$\frac{\Gamma \vdash A \cong B : s \qquad \Gamma \vdash t \cong u : A \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \text{neutral}\ (\texttt{cast}\ A\ B\ e\ t) \qquad \text{neutral}\, u}{\Gamma \vdash \texttt{cast}\ A\ B\ e\ t \cong_{ne} u : B}\ \textsc{Cast-refl-L}$$

$$\frac{\Gamma \vdash B \cong A : s \qquad \Gamma \vdash t \cong u : A \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \text{neutral}\, t \qquad \text{neutral}\ (\texttt{cast}\ A\ B\ e\ u)}{\Gamma \vdash t \cong_{ne} \texttt{cast}\ A\ B\ e\ u : A}\ \textsc{Cast-refl-R}$$

$$\frac{\Gamma \vdash t, u : A : \mathcal{U}_i \qquad \text{whnf}\, A \qquad \Gamma \vdash t \cong^{\downarrow}_{ne} u : A}{\Gamma \vdash t \cong u : A}\ \textsc{Ne-whnf}$$

$$\frac{\Gamma \vdash A \Rightarrow^{*} B : \mathcal{U}_i \qquad \text{whnf}\ B \qquad \Gamma \vdash t \cong_{ne} u : A}{\Gamma \vdash t \cong^{\downarrow}_{ne} u : B}\ \textsc{Ne-Red}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \Rightarrow^{*} B : \mathcal{U}_i \qquad \Gamma \vdash t \Rightarrow^{*} t' : A \qquad \Gamma \vdash u \Rightarrow^{*} u' : A \\ \text{whnf}\ B, \text{whnf}\ t', \text{whnf}\ u' \qquad \Gamma \vdash t' \cong u' : B\end{array}}{\Gamma \vdash t \cong^{\downarrow} u : A}\ \textsc{Whnf-Red}$$

Fig. 6: $\mathsf{CIC}^{\text{obs}}$ Algorithmic Conversion Rules (except congruence rules) *[ConversionGen.agda]*

provide a canonical typing derivation by splitting the typing judgment into two: one judgment that infers the type of a term and an other one that checks that the inferred type of a term is convertible to the type given as input. This allows bidirectional typing to restrict the use of the conversion rule only to well-controlled places, and thus to provide only canonical derivations. In this presentation, it should be noticed that neutral terms infers an arbitrary terms (for instance, the application rule infers the type of the codomain of the function with an additional substitution) whereas other weak-head normal forms always infer a type also in weak-head normal form. But the structural rules for conversion correspond to a relational version of the type judgments, so that in some sense conversion subsumes typing. This means that we need to reflect this important distinction in the algorithmic conversion because the structural conversion rules for neutral terms ($\Gamma \vdash t \cong_{ne} u : A$) will naturally be performed at an arbitrary type A whereas $\Gamma \vdash t \cong u : A$ is always done at a type A in weak-head normal form.

Because conversion of whnf must contain conversion of neutrals as a particular case, we need those two notions to be compatible. To that end, we introduce two other judgments: $\Gamma \vdash t \cong^{\downarrow}_{ne} u : B$ means that $\Gamma \vdash t \cong_{ne} u : A$ and B is the whnf of A (Rule Ne-Red) and $\Gamma \vdash t \cong^{\downarrow} u : A$ means that $\Gamma \vdash t' \cong^{\downarrow} u' : A'$ and t', u' and B are the whnf of t, u and A respectively (Rule Whnf-Red).

Let us now turn to the description of the relation $\Gamma \vdash t \cong u : A$ which mainly contains congruence rules for weak-head constructors, that are used in particular to show that reflexivity is admissible. Those congruence rules just ask for convertibility of each sub-argument, with some sanity conditions on the leaves, to ensure that only well-typed terms are considered in the conversion relation. Then, the rule Ne-whnf says that two neutral terms are comparable as whnf when they are comparable as neutral terms.

The relation $\Gamma \vdash t \cong_{ne} u : A$ contains a first rule to deal with proof-irrelevance in $\mathbf{\Omega}$ (Rule Proof-irr). As any term in $\mathbf{\Omega}$ is neutral, this rule only checks that the two terms are proofs of the same proposition. The rule for variables (Rule Var-refl) applies when there is the same variable x on the left and on the right, and this variable is declared in the local context Γ.

Then, there are four congruence rules to deal with eliminators. An eliminator is neutral when one of its scrutinees is neutral.The situation for `cast` is more complex as there are three different scrutinees (the two types and the term to be cast) and the whole term is neutral as soon as one of them is neutral. There is also a last kind of neutrals for `cast` which corresponds to impossible casts, that is casts between types with different head constructors. We can actually factorize all those cases and present only one rule (Cast-cong) that simply asks both casts to be neutral terms, at the price of a seemingly less accurate system. Indeed, because we are oblivious to the reason why the casts are neutral, all preconditions are asking for conversion as weak-head normal form instead of specializing in the case of neutral terms. However, by inversion on the rule, it is possible to show that two neutral terms are convertible as whnf if and only if they are convertible as neutral terms, so in the end this factorized rule is equivalent to a system with one rule per kind of neutral terms as defined in *[Conversion.agda]*.

To deal with Cast-Refl, we need to introduce two rules, one for simplification of cast on the left, and one on the right. This is because we have no rule for symmetry (to keep the system algorithmic) and symmetry must be an admissible rule. So the conversion rule is split into the two rules Cast-refl-L and Cast-refl-R. Again, we use a factorization to get only two rules, not specializing on the reason why a cast is neutral.

The main point of this algorithmic conversion is that it does not contain any rule for symmetry or transitivity. This is because they make it very difficult to prove decidability of conversion. However, we can show that symmetry (*[Symmetry.agda]*) and transitivity are admissible (*[Transitivity.agda]*).

4.3 Decidability of Algorithmic Conversion

We now turn to the definition of a decision procedure for the algorithmic conversion *[Decidable.agda]*. Actually, what we first prove is the decidability of algorithmic conversion for two terms t and u, assuming that we know that $\Gamma \vdash t \cong_{ne} t : A$ and $\Gamma \vdash u \cong_{ne} u : A$. The fact that algorithmic conversion is reflexive is actually a consequence of the completeness of algorithmic conversion with respect to declarative conversion that will be shown in the next section. The hypothesis that t and u are in diagonal of the algorithmic conversion contains a lot of information, because by inversion on the derivations, we can actually recover the fact that t and u can be reduced to a whnf whose subterms can also be reduced in whnf, and this again and again up-to getting a deep normal form.

The decidability proof of conversion for MLTT in [2] coarsely amounts to zipping the two reflexivity proofs together, showing that when the two derivations do not share the exact same structure, then the two terms are not convertible. This is not the case anymore in presence of the rules Cast-refl-L and Cast-refl-R and the reasoning cannot stay on the "diagonal" of the algorithmic conversion. This is not an issue as actually from the fact that $\Gamma \vdash t \cong_{ne} t' : A$, we can deduce that both t and t' can be put in deep normal form and so somehow, $\Gamma \vdash t \cong_{ne} t' : A$ can be used as termination witness in the same way as $\Gamma \vdash t \cong_{ne} t : A$.

However, the main difficulty in this new setting is that it is not true anymore that when the two derivations do not share the exact same structure, then the two terms are not convertible. Consider for instance $\texttt{cast}\ A\ B\ e\ t$ against t: the reflexivity proofs for these two terms cannot share the same structure, yet they are convertible by Rule Cast-refl-L. In addition, in the more complex case of $\texttt{cast}\ A\ B\ e\ t$ against $\texttt{cast}\ A'\ B'\ e'\ t'$, there are three cases to consider, because the last rule to show that they are convertible can be either Cast-cong, Cast-refl-L or Cast-refl-R. This means in particular that the proof that two terms are algorithmically convertible is not unique anymore, and the decidability procedure has to do an arbitrary choice, depending on which order it tests the three different possibility and backtracks.

The statement of decidability needs to be generalized in the following way.

Theorem 1 (Decidability of algorithmic conversion *[Decidable.agda]***).** *For any natural number n, given two proofs of neutral comparison $\pi : \Gamma \vdash t \cong_{ne} t' : A$ and $\pi' : \Delta \vdash u \cong_{ne} u' : B$ such that $\vdash \Gamma \equiv \Delta$ and $\texttt{size}(\pi) + \texttt{size}(\pi') < n$, knowing whether there exists a type C such that $\Gamma \vdash t \cong_{ne} u : C$ is decidable.*

Note that the statement is based on a notion of size of a derivation, noted size, because the algorithm does recursive calls that are not structurally decreasing. To conclude on the completeness of algorithmic conversion *[Completeness.agda]*, we reuse the logical relation setting described in [2] for proving strong normalization and decidability of conversion in various type theories, later extended in [22,23]. We do not detail the definition of the logical relation here as there is not specific to our system, what is important is it provides the following consequence.

5 Inductive Definitions

On top of the rules from Section 3, $\mathsf{CIC}^{\text{obs}}$ includes a scheme to define proof-relevant inductive types that is based on the scheme of CIC (as defined in [26]). Inductive definitions are not manipulated as first class objects: instead, the user declares all the necessary inductive types using a standard syntax, before starting their proof. After each declaration, the theory is automatically extended with the new type former, inductive constructors, etc.

The syntax for the inductive scheme of $\mathsf{CIC}^{\text{obs}}$ is exactly the same as the scheme of CIC; the difference lies in the fact that inductive definitions will additionally have to generate projections for the observational equality types and computation rules for the `cast` operator. We start by explaining how it works for inductive types without indices, and then we extend it to general indexed inductive definitions by using the Fordism transformation and some syntactic sugar. We will spare the reader the added complexity of mutually defined families, which is mathematically direct but heavy on notation.

5.1 Inductive Definitions Without Indices

We use a syntax based on the one used by the Coq proof assistant for inductive definitions. The general form of a non-indexed type looks like this:

$\texttt{Inductive Ind}\ (\vec{a} : \vec{A}) : \mathcal{U}_\ell :=$
$\mid c_0 : \forall\ (\vec{b} : \vec{B}_0),\ \texttt{Ind}\ \vec{a}$
$\mid \ldots$
$\mid c_n : \forall\ (\vec{b} : \vec{B}_n),\ \texttt{Ind}\ \vec{a}$

In order to represent arbitrary contexts of parameters more compactly, we used a vector notation. The parameter $(\vec{a} : \vec{A})$ represents a context of the form $a_1 : A_1, \ldots, a_m : A_m$ where each type may depend on the previous ones. Similarly, every constructor of the inductive type has a context of arguments, that may include recursive calls to `Ind` in *strictly positive positions*—however we will not be paying special attention to recursive calls, as their treatment is not affected by the observational equality. The universe $\mathcal{U}_\ell$ must be larger than all the types that appear in the constructor arguments $\vec{B}_i$. Inductive definitions in the sort of propositions $\boldsymbol{\Omega}$ are not allowed.

After the user makes such a definition, the system is extended with the new type former `Ind` and the inductive constructors c_0, ... c_n with their prescribed types. Additionally, $\mathsf{CIC}^{\text{obs}}$ provides two operators `match` and `fix` that are used to define functions out of an inductive definition, following the typing and computation rules described by the [11]. As we explain in Section 2, this elimination principle is enough to completely determine the observational equality between any two inhabitants of `Ind`, thus our system does not provide any additional rule for this. However, the observational equality between two instances of `Ind` does not benefit from any such principle, so we add "projection" operators to characterize equalities between inductive types:

$$\texttt{eq\_}c_i : \forall\ (\vec{a} : \vec{A})\ (\vec{a}' : \vec{A}),\ \texttt{Ind}\ \vec{a} \sim \texttt{Ind}\ \vec{a}' \to \vec{B}_i[\vec{a}] \sim \vec{B}_i[\vec{a}'] \qquad (\forall\ \texttt{i})$$

The projections $\texttt{eq\_}c_i$ are generated when the user makes the definition of `Ind`, just like the constructors c_i. Remark that the codomains of these projections are equalities between two vectors, which is a notational shorthand for a vector of equalities. In practice, this means that each $\texttt{eq\_}c_i$ will be implemented as a family of projections $(\texttt{eq\_}c_{i,j})$, where each projection depends on the previous ones. Thus, we get as many projections as there are constructor arguments in the inductive definition. Finally, we add computation rules for cast:

$$\texttt{cast}\ (\texttt{Ind}\ \vec{a})\ (\texttt{Ind}\ \vec{a}')\ \texttt{e}\ (c_i\ \vec{b})\ \equiv\ c_i\ (\texttt{cast}\ (\vec{B}_i[\vec{a}])\ (\vec{B}_i[\vec{a}'])\ (\texttt{eq\_}c_i\ \vec{a}\ \vec{a}'\ \texttt{e})\ \vec{b})\quad (\forall\ \texttt{i})$$

5.2 Deriving a Scheme for Indexed Inductive Types

In order for $\mathsf{CIC}^{\mathrm{obs}}$ to be a proper extension of CIC, we need to extend our scheme to indexed inductive definitions. These get a bit messier than non-indexed definitions, but in fact we already have all the pieces we need: as we saw in Section 2.2, the rule Cast-Refl allows us to use the Fordism transformation and *faithfully* encode indexed inductive types with parametrized inductive types. Consequently, we will define the scheme for indexed definitions in terms of the scheme for non-indexed definitions, using syntactic sugar and elaboration. That way, the typing and computation rules of CIC that involve indexed inductive types remain valid in $\mathsf{CIC}^{\mathrm{obs}}$, but the inductive types and constructors are elaborated to their non-indexed counterpart under the hood.

We now explain in detail how this elaboration process works. When the user defines an indexed inductive type Ind, they are actually defining the forded version $\texttt{Ind}_{\mathbb{F}}$ *via* the scheme for non-indexed definitions:

$$\begin{array}{ll}
\texttt{Inductive Ind}\ (\vec{a} : \vec{A}) : \forall\ (\vec{x} : \vec{X}), \mathcal{U}_\ell := & \texttt{Inductive Ind}_{\mathbb{F}}\ (\vec{a} : \vec{A})\ (\vec{x} : \vec{X}) : \mathcal{U}_\ell := \\
\mid c_0\ : \forall\ (\vec{b} : \vec{B_0}),\ \texttt{Ind}\ \vec{a}\ \vec{y_0} & \mid c_{0\mathbb{F}}\ : \forall\ (\vec{b} : \vec{B_0}),\ \vec{y_0} \sim \vec{x} \to\ \texttt{Ind}_{\mathbb{F}}\ \vec{a}\ \vec{x} \\
\mid \ldots & \mid \ldots \\
\mid c_n\ : \forall\ (\vec{b} : \vec{B_n}),\ \texttt{Ind}\ \vec{a}\ \vec{y_n} & \mid c_{n\mathbb{F}}\ : \forall\ (\vec{b} : \vec{B_n}),\ \vec{y_n} \sim \vec{x} \to\ \texttt{Ind}_{\mathbb{F}}\ \vec{a}\ \vec{x}
\end{array}$$

The scheme generates projections for observational equalities between the constructor arguments, *including* the index equalities $\vec{y_i} \sim \vec{x}$ that are hidden in the user definition. Then, our system defines Ind and its constructors in terms of their forded counterparts:

$$\texttt{Ind}\ \vec{a}\ \vec{x}\ \equiv\ \texttt{Ind}_{\mathbb{F}}\ \vec{a}\ \vec{x} \qquad\qquad c_i\ \vec{b}\ \equiv c_{i\mathbb{F}}\ \vec{b}\ \texttt{refl}$$

The pattern matching on inhabitants of the indexed inductive type is elaborated to a pattern matching on the forded version, by inserting a cast in each branch. Concretely, consider the following pattern matching on $\texttt{i} : \texttt{Ind}\ \vec{a}\ \vec{x}$:

$$\texttt{match i return P with} \mid c_0\ \vec{b} \Rightarrow t_0\ \ \mid ....\ \mid c_n\ \vec{b} \Rightarrow t_n\ \texttt{end}$$

The return type is $\texttt{P}\ \vec{x}\ \texttt{i}$, and thus in the branch for $c_i\ \vec{b}$, the term t_i provided by the user has type $\texttt{P}\ \vec{y_i}\ (c_i\ \vec{b})$. After the elaboration, this branch matches a forded pattern $c_{i\mathbb{F}}\ \vec{b}\ e$, and should now return a result of type $\texttt{P}\ \vec{x_i}\ (c_{i\mathbb{F}}\ \vec{b}\ e)$. We can obtain this result by type-casting the user-supplied term t_i along the equality proof e to obtain

$$\texttt{cast}\ (\texttt{P}\ \vec{y_i}\ (c_i\ \vec{b}))\ (\texttt{P}\ \vec{x_i}\ (c_{i\mathbb{F}}\ \vec{b}\ e))\ (\texttt{ap2 P}\ (c_{i\mathbb{F}}\ \vec{b}\ e))\ t_i$$

where ap2 is a slight generalization of the proof that function applications preserve equalities. Thanks to the rule Cast-Refl, this elaboration preserves the computation rule of the pattern-matching for indexed inductive types. Note that there is nothing special to do for fixpoints, they work out of the box. This concludes the description of our formal system $\mathsf{CIC}^{\text{obs}}$.

6 Consistency of the Theory

In Section 2 we saw that combining the inductive scheme of CIC with the observational equality can endanger the consistency of the theory if we are not careful. In the end, it is possible to fix the issue by picking a better definition for the observational equality of inductive types, but now we want to make sure that this new definition will not lead to another inconsistency. To do this, we build a model of $\mathsf{CIC}^{\text{obs}}$ in set theory, thereby reducing the consistency of our system to the consistency of ZFC set theory with Grothendieck universes. Our model is mostly an extension of the one that was presented in [23] to general inductive definitions, using the interpretation of inductive definitions that was developed in [26].

6.1 Observational Type Theory in Sets

We work in ZFC set theory with a countable hierarchy of Grothendieck universes $\mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2$, *etc.* We write $\mathbf{\Omega} := \{\bot, \top\}$ for the lattice of truth values, and given $p \in \mathbf{\Omega}$ we write $\mathsf{val}\ p$ for the associated set $\{x \in \{*\} \mid p\}$. Since our goal is to interpret a dependent type theory, we will need set-theoretic dependent products and dependent sums. We write the former as $(a \in A) \to (B\ a)$, and the latter as $(a \in A) \times (B\ a)$ to distinguish them from their type-theoretic counterparts.

Our model will be based on the *types-as-sets* interpretation of dependent type theory [12], according to which contexts are interpreted as sets, types and terms over a context Γ become sets indexed over the interpretation of Γ, the typing relation corresponds to set membership, and conversion is interpreted as the set-theoretic equality. Such models have already been defined for a wide variety of type theories; of particular interest to us is the model of [26] which supports an impredicative sort of propositions (interpreted as the lattice of truth values) and the full scheme of inductive definitions of CIC. Since ZFC set theory is extensional by nature, this model also validates the principles of function extensionality and proposition extensionality, which would *almost* make it a model of $\mathsf{CIC}^{\text{obs}}$, were it not for two small issues.

The first issue is the absence of the observational equality and the `cast` operator in the model of [26]. We can easily fix this by interpreting the observational equality as the set-theoretic equality, and `cast` as the identity function. That way, `cast` verifies all the desired equations for trivial reasons, including the rule Cast-Refl. After all, the model does not differentiate between conversion and propositional equality!

The second issue is a bit more serious, and deals with the universes. In [26], the authors directly interpret the type-theoretic universes as the corresponding Grothendieck universes, which is perfectly fine for CIC. But this does not work for $\mathsf{CIC}^{\text{obs}}$, as we would lose the injectivity of dependent products: consider for

$$\begin{array}{ll}
[\![\Gamma \vdash \mathcal{U}_j]\!]_\rho & := \langle\ \mathbf{V}_j \times \mathbf{V}_j\ ,\ \emptyset \,\rangle \\
[\![\Gamma \vdash \Omega]\!]_\rho & := \langle\ \Omega\ ,\ \emptyset \,\rangle \\
[\![\Gamma \vdash \Pi^{\mathcal{U}_j,\mathcal{U}_k} (x : A).\, B]\!]_\rho & := \langle\ (x \in \mathsf{fst}\ [\![\Gamma \vdash A]\!]_\rho) \to \mathsf{fst}\ [\![\Gamma, A \vdash B]\!]_{\rho,x}\ , \\
& \qquad ([\![\Gamma \vdash A]\!]_\rho\ ,\ \lambda x\ .\ [\![\Gamma, A \vdash B]\!]_{\rho,x})\,\rangle \\
[\![\Gamma \vdash \Pi^{\Omega,\mathcal{U}_j} (x : A).\, B]\!]_\rho & := \langle\ (x \in \mathsf{val}\ [\![\Gamma \vdash A]\!]_\rho) \to \mathsf{fst}\ [\![\Gamma, A \vdash B]\!]_{\rho,x}\ , \\
& \qquad (\mathsf{val}\ [\![\Gamma \vdash A]\!]_\rho\ ,\ \lambda x\ .\ [\![\Gamma, A \vdash B]\!]_{\rho,x})\,\rangle \\
[\![\Gamma \vdash \mathtt{Ind}\ \vec{X}]\!]_\rho & := \langle\ \mathsf{IndElem}\ [\![\Gamma \vdash \vec{X}]\!]_\rho\ ,\ \mathsf{IndLabel}\ [\![\Gamma \vdash \vec{X}]\!]_\rho \,\rangle
\end{array}$$

Fig. 7: Codes for universes, dependent products and inductive types

instance the two types $\mathtt{Empty} \to \mathbb{N}$ and $\mathtt{Empty} \to \mathbb{B}$. Both are interpreted as a singleton set, but in $\mathsf{CIC}^{\mathrm{obs}}$ we can prove that they cannot be equal. To recover this injectivity, we *label* the sets in the universe with additional information that indicates how they were built. This way, the type $\mathtt{Empty} \to \mathbb{N}$ is interpreted as a singleton set *and* an indication that it is a function type from Empty to $\mathbb{N}$, while $\mathtt{Empty} \to \mathbb{B}$ has a different label.

6.2 Coinductive Labels for Inductive Types

In this section, we give a proper definition for our labelled universe. The technique of using labels to build a universe that is generic for sets *and* ensures the injectivity of dependent products is a re-reading of the technique of [13]. However, his construction seems difficult to extend with parametrized inductive types—the use of induction-recursion seems to force us to have the injectivity of parameters, which we do not want (cf Section 2.3). Therefore we ditch induction-recursion for a definition that is somewhat more set-theoretic: our interpretation of the universe $\mathcal{U}_i$ is simply $\mathbf{V}_i \times \mathbf{V}_i$, meaning that a code in the universe is a pair of sets. The first set of the pair is the (semantic) type, and the second set is the label. The "El" function that transforms a code into a type is thus simply the first projection.

Fig. 7 shows the interpretation for the proof-relevant type formers of $\mathsf{CIC}^{\mathrm{obs}}$. The interpretation function that transforms a syntactic object into a semantic object is written $[\![\Gamma \vdash \_]\!]_\rho$, where ρ is a set-theoretic function that assigns a set to every variable of the context Γ. Unsurprisingly, the syntactic universes $\mathcal{U}_i$ and Ω are interpreted as their semantic counterparts, with the default label (the empty set). Dependent products also are interpreted as their set-theoretic counterparts, but in that case the label contains the domain and the codomain, ensuring that two dependent products are not identified unless their domain and codomain are themselves equal.

The case of the inductive definitions is a bit more involved. Thankfully we do not need to treat indices, as they are encoded using Fordism (cf Section 5.2). Thus, we consider a non-indexed inductive definition Ind as in Section 5, with a vector of parameters $\vec{A}$:

```
Inductive Ind (a⃗ : A⃗) : 𝒰_ℓ :=
| c₀ : ∀ (b⃗ : B⃗₀), Ind a⃗
| ...
| cₙ : ∀ (b⃗ : B⃗ₙ), Ind a⃗
```

Given any vector $\vec{X}$ of elements of the family of sets $\mathsf{fst}(\llbracket\ \vec{A}\ \rrbracket_\rho)$, we define $\mathsf{IndElem}\ \vec{X}$ to be the set constructed in [26], which is well-defined if the definition of `Ind` is strictly positive and all the the interpretations of the $\vec{B_i}$ are well-defined. Reproducing their construction in full detail would take us too far from the scope of this paper, so we will simply mention that it is the initial algebra of the set-theoretic endofunctor corresponding to `Ind` evaluated in $\vec{X}$. This gives us the first projection of $\llbracket\ \Gamma \vdash \texttt{Ind}\ \vec{X}\ \rrbracket_\rho$, and now we need to define the second projection $\mathsf{IndLabel}\ \vec{X}$. Recall from Section 2.3 that we would like the equality of two instances of `Ind` to satisfy:

$$\texttt{Ind}\ \vec{X} \sim \texttt{Ind}\ \vec{Y} \quad \longleftrightarrow \quad (\vec{B_0}(\vec{X}), ..., \vec{B_n}(\vec{X})) \sim (\vec{B_0}(\vec{Y}), ..., \vec{B_n}(\vec{Y})).$$

In other words, `Ind` should be determined up to equality by the types of its constructor arguments. Therefore, it is natural to define its label directly as the list of these types:

$$\mathsf{IndLabel}\ \vec{X} \quad = \quad (\llbracket\Gamma, \vec{A} \vdash \vec{B_0}\rrbracket_{(\rho,\vec{X})}, ..., \llbracket\Gamma, \vec{A} \vdash \vec{B_n}\rrbracket_{(\rho,\vec{X})}).$$

However, remark that $\vec{B_i}$ may contain a recursive call to `Ind`, whose interpretation is defined using $\mathsf{IndLabel}$, so this definition is really an equation that we need to solve. Fortunately, a simple look at the shape of that equation reveals that it is in fact a definition for an infinite tree whose nodes are labeled with sets, which we take as our solution. Note that the result is indeed an inhabitant of $\mathbf{V}_\ell$, since the sets that intervene in its construction (the interpretation of the types of the constructor arguments and their labels) are all in $\mathbf{V}_\ell$. With this definition of $\mathsf{IndLabel}$, we get the following property:

Lemma 1. *If the inductive definition* `Ind` *is strictly positive,* $\llbracket\Gamma \vdash \vec{X}\rrbracket_\rho$ *is well-defined, and all the* $\llbracket\Gamma, \vec{A} \vdash \vec{B_i}\rrbracket_{(\rho,\vec{X})}$ *are well-defined, then* $\llbracket\Gamma \vdash \texttt{Ind}\ \vec{X}\rrbracket_\rho$ *is well-defined. Furthermore,* $\llbracket\Gamma \vdash \texttt{Ind}\ \vec{X}\rrbracket_\rho = \llbracket\Gamma \vdash \texttt{Ind}\ \vec{Y}\rrbracket_\rho$ *is equivalent to*

$$\forall i, \quad \llbracket\Gamma, \vec{A} \vdash \vec{B_i}\rrbracket_{(\rho,\llbracket\Gamma\vdash\vec{X}\rrbracket_\rho)} = \llbracket\Gamma, \vec{A} \vdash \vec{B_i}\rrbracket_{(\rho,\llbracket\Gamma\vdash\vec{Y}\rrbracket_\rho)}.$$

6.3 Soundness of the Model

The definition of the observational universe is the only new insight of our construction; the rest follows the strategy laid out in [26]. For the sake of completeness, we give an outline of the definition and of the proof of soundness in this section.

Ultimately, our model is defined in terms of *partial* functions from the syntax to the semantics. We use a function $\llbracket\_\rrbracket$ that interprets contexts as sets and a function $\llbracket\Gamma \vdash \_\rrbracket_\rho$ that interprets terms and types in context Γ as sets indexed by $\rho \in \llbracket\Gamma\rrbracket$ (Fig. 8). Both functions are mutually defined by recursion on the

$$\begin{aligned}
[\![\bullet]\!] &:= \{\emptyset\} \\
[\![\Gamma, x : A : \mathcal{U}_i]\!] &:= \{(\rho, a) \mid \rho \in [\![\Gamma]\!] \ \wedge\ a \in \mathsf{fst}\ [\![\Gamma \vdash A]\!]_\rho\} \\
[\![\Gamma, x : A : \Omega]\!] &:= \{(\rho, a) \mid \rho \in [\![\Gamma]\!] \ \wedge\ a \in \mathsf{val}\ [\![\Gamma \vdash A]\!]_\rho\}
\end{aligned}$$

$$\begin{aligned}
[\![\Gamma \vdash x]\!]_\rho &:= \rho(x) \\
[\![\Gamma \vdash \lambda(x : F).\, t]\!]_\rho &:= (x \in \mathsf{fst}\ [\![\Gamma \vdash F]\!]_\rho) \mapsto ([\![\Gamma, F \vdash t]\!]_{\rho,x}) \\
[\![\Gamma \vdash t\ u]\!]_\rho &:= [\![\Gamma \vdash t]\!]_\rho([\![\Gamma \vdash u]\!]_\rho)
\end{aligned}$$

$$\left.\begin{aligned}
[\![\Gamma \vdash c_i\ \vec{b}]\!]_\rho &:= \\
[\![\Gamma \vdash \texttt{match t return P with } \{c_i\ \vec{b} \Rightarrow t_i\}]\!]_\rho &:= \\
[\![\Gamma \vdash \texttt{fix}\ f\ \vec{x} := t]\!]_\rho &:=
\end{aligned}\right\} \text{(as in Lee \textit{et al.})}$$

$$\begin{aligned}
[\![\Gamma \vdash \bot]\!]_\rho &:= \bot \\
[\![\Gamma \vdash \bot\texttt{-elim}\ A\ t]\!]_\rho &:= \text{undefined} \\
[\![\Gamma \vdash t \sim_A u]\!]_\rho &:= \top \text{ if } [\![\Gamma \vdash t]\!]_\rho = [\![\Gamma \vdash u]\!]_\rho \\
& \ \bot \text{ otherwise} \\
[\![\Gamma \vdash \texttt{cast}\ A\ B\ e\ t]\!]_\rho &:= [\![\Gamma \vdash t]\!]_\rho \\
[\![\Gamma \vdash \Pi^{\mathcal{U}_j, \Omega}(y : A).\, B]\!]_\rho &:= \forall x \in (\mathsf{fst}\ [\![\Gamma \vdash A]\!]_\rho),\ [\![\Gamma, A \vdash B]\!]_{\rho,x} \\
[\![\Gamma \vdash \Pi^{\Omega, \Omega}(y : A).\, B]\!]_\rho &:= \forall x \in (\mathsf{val}\ [\![\Gamma \vdash A]\!]_\rho),\ [\![\Gamma, A \vdash B]\!]_{\rho,x}
\end{aligned}$$

Fig. 8: Interpretation of contexts and proof-relevant terms of $\mathsf{CIC}^{\text{obs}}$

raw syntax, and we will then prove that they are total on well-typed terms by induction on the typing derivations. Variables, lambda-abstractions and applications are interpreted respectively as projections from the context, set-theoretic functions and applications. In order to interpret the inductive constructors and the `match` and `fix` operators, we need to develop a proper theory of set-theoretic induction. Since this part is completely orthogonal to the observational primitives, we deem it out of the scope of this work and we refer the interested reader to the literature instead. In [26], the authors use induction principles instead of `match` and `fix`, but argue that the two are equivalent. A model directly based on the latter can be found in [14]. The $\bot$ proposition is interpreted as the false proposition of ZFC, the observational equality as the equality of ZFC, and the cast operator as the identity function. Finally, the proof-irrelevant dependent products are interpreted as set-theoretic quantifications. The proofs of propositions such as `transport` or Π^1_ϵ do not need to be interpreted—after all, the model is proof-irrelevant.

In order to prove the soundness of our interpretation, we need to extend it to weakenings and substitutions between contexts. Assume Γ and Δ are syntactical contexts, and A and t are syntactical terms. In case $[\![\Gamma, x : A : s, \Delta]\!]$ and $[\![\Gamma, \Delta]\!]$ are well-defined, let π_A be the projection:

$$\pi_A : [\![\Gamma, x : A : s, \Delta]\!] \to [\![\Gamma, \Delta]\!] \qquad (\vec{x_\Gamma}, x_A, \vec{x_\Delta}) \mapsto (\vec{x_\Gamma}, \vec{x_\Delta}).$$

In case $[\![\Gamma, \Delta[x := t]]\!]$ and $[\![\Gamma, x : A : s, \Delta]\!]$ are well-defined, we define the function σ_t by:

$$\sigma_t : [\![\Gamma, \Delta[x := t]]\!] \to [\![\Gamma, x : A : s, \Delta]\!] \qquad (\vec{x_\Gamma}, \vec{x_\Delta}) \mapsto (\vec{x_\Gamma}, [\![\Gamma \vdash t]\!]_{\vec{x_\Gamma}}, \vec{x_\Delta}).$$

Theorem 2 (Soundness of the Standard Model).

1. *If* $\vdash \Gamma$ *then* $[\![\Gamma]\!]$ *is defined.*
2. *If* $\Gamma \vdash A : \Omega$ *then* $[\![\Gamma \vdash A]\!]_\rho$ *is a semantic proposition for all* $\rho \in [\![\Gamma]\!]$.
3. *If* $\Gamma \vdash A : \mathcal{U}_i$ *then* $[\![\Gamma \vdash A]\!]_\rho$ *is in* $\mathbf{V}_i$ *for all* $\rho \in [\![\Gamma]\!]$.
4. *If* $\Gamma \vdash t : A : \Omega$ *then* $[\![\Gamma \vdash t]\!]_\rho \in \mathsf{val}([\![\Gamma \vdash A]\!]_\rho)$ *for all* $\rho \in [\![\Gamma]\!]$.
5. *If* $\Gamma \vdash t : A : \mathcal{U}_i$ *then* $[\![\Gamma \vdash t]\!]_\rho \in \mathsf{fst}([\![\Gamma \vdash A]\!]_\rho)$ *for all* $\rho \in [\![\Gamma]\!]$.
6. *If* $\Gamma \vdash t \equiv u : A$ *then* $[\![\Gamma \vdash t]\!]_\rho = [\![\Gamma \vdash u]\!]_\rho$ *for all* $\rho \in [\![\Gamma]\!]$.

Since our model interprets the false proposition $\bot$ as the empty set, we get a proof of consistency:

Theorem 3 (Consistency). *There are no proofs of* $\bot$ *in the empty context.*

Furthermore, by inspecting the normal forms provided by the normalization theorem, we note that the only neutral terms in the empty context are stuck casts. But having a stuck `cast` requires an equality proof between two incompatible types, which cannot exist from our definition of the universe. From there, we derive a canonicity theorem for inductive types: all elements of an inductive type without indices reduce to canonical elements in the empty context.

7 Conclusion and Future Work

We proposed a systematic integration of indexed inductive types with an observational equality, by defining a notion of observational equality that satisfies the computational rule of Martin-Löf's identity type and by using Fordism, a general technique to faithfully encode indexed inductive types with non-indexed types and equality. We developed a formal proof that this additional computation rule, although not present in previous works on observational equality, can be integrated to the system without compromising the decidability of conversion. This extension of CIC with an observational equality has been implemented at the top of the Coq proof assistant by using the recently introduced rewrite rules.

Although the technique has been developed in the setting of CIC and Coq specifically, there is no obstacle to adapt it to other settings such as Lean or Agda. Adaption to Lean should be pretty straightforward as it is sharing most of its metatheory with Coq. A partial version of $\mathsf{CIC}^{\mathrm{obs}}$ could be provided in Agda with rewrite rules. However, the management of elimination of inductive types in Agda is not done using an explicit pattern-matching syntax à la Coq, for which we can define new reduction rules. Instead, functions on inductive types are defined using case splitting trees and an exhaustivity checker. Therefore, a proper treatment of $\mathsf{CIC}^{\mathrm{obs}}$ in Agda would require modifications of the case splitting engine, similarly to what has been done for Cubical Agda [27].

8 Data-Availability Statement

The Agda companion formalization is available both on GitHub and as a long-term archived artifact [24].

References

1. Abel, A., Coquand, T.: Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality. Logical Methods in Computer Science **Volume 16, Issue 2** (2020). `https://doi.org/10.23638/LMCS-16(2:14)2020`. URL `https://lmcs.episciences.org/6606`
2. Abel, A., Öhman, J., Vezzosi, A.: Decidability of conversion for type theory in type theory. Proceedings of the ACM on Programming Languages **2**(POPL), 23:1–23:29 (2018). `https://doi.org/10.1145/3158111`. URL `http://doi.acm.org/10.1145/3158111`
3. Adjedj, A., Lennon-Bertrand, M., Maillard, K., Pédrot, P.M., Pujet, L.: Martin-löf à la coq (2023)
4. Allais, G., McBride, C., Boutillier, P.: New equations for neutral terms: A sound and complete decision procedure, formalized. In: Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming, DTP '13, pp. 13–24. ACM, New York, NY, USA (2013). `https://doi.org/10.1145/2502409.2502411`. URL `http://doi.acm.org/10.1145/2502409.2502411`
5. Altenkirch, T., McBride, C.: Towards Observational Type Theory (2006). URL `http://www.strictlypositive.org/ott.pdf`
6. Altenkirch, T., McBride, C., Swierstra, W.: Observational equality, now! In: Proceedings of the Workshop on Programming Languages meets Program Verification (PLPV 2007), pp. 57–68 (2007). `https://doi.org/10.1145/1292597.1292608`
7. Atkey, B.: Simplified Observational Type Theory (2017). URL `https://github.com/bobatkey/sott`
8. Barendregt, H.P.: Lambda calculi with types. In: Handbook of logic in computer science (vol. 2) background: computational structures, pp. 117–309 (1993)
9. Berg, B.v.d., Garner, R.: Types are weak ω-groupoids. Proceedings of the London Mathematical Society **102** (2008)
10. Cavallo, E., Harper, R.: Higher inductive types in cubical computational type theory. Proc. ACM Program. Lang. **3**(POPL) (2019). `https://doi.org/10.1145/3290314`. URL `https://doi.org/10.1145/3290314`
11. Coq Development Team, T.: The Coq proof assistant reference manual (2016). URL `http://coq.inria.fr`. Version 8.6
12. Dybjer, P.: Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics, p. 280–306. Cambridge University Press, USA (1991)
13. Gratzer, D.: An inductive-recursive universe generic for small families (2022). `https://doi.org/10.48550/ARXIV.2202.05529`. URL `https://arxiv.org/abs/2202.05529`
14. Lee, G., Werner, B.: Proof-irrelevant model of CC with predicative induction and judgmental equality. Logical Methods in Computer Science **Volume 7, Issue 4** (2011). `https://doi.org/10.2168/lmcs-7(4:5)2011`
15. Lennon-Bertrand, M.: Complete Bidirectional Typing for the Calculus of Inductive Constructions. In: L. Cohen, C. Kaliszyk (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 193. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021). `https://doi.org/10.4230/LIPIcs.ITP.2021.24`. URL `https://drops.dagstuhl.de/opus/volltexte/2021/13919`
16. Lennon-Bertrand, M.: Bidirectional Typing for the Calculus of Inductive Constructions. Theses, Nantes Université (2022). URL `https://theses.hal.science/tel-03848595`

17. Martin-Löf, P.: An intuitionistic theory of types: Predicative part. In: H. Rose, J. Shepherdson (eds.) Logic Colloquium '73, *Studies in Logic and the Foundations of Mathematics*, vol. 80, pp. 73 – 118. Elsevier (1975). https://doi.org/https://doi.org/10.1016/S0049-237X(08)71945-1. URL http://www.sciencedirect.com/science/article/pii/S0049237X08719451
18. McBride, C.: Dependently typed functional programs and their proofs. Ph.D. thesis, University of Edinburgh (2000)
19. McBride, C.: Hier soir, an ott hierarchy. Blog post (2011). URL https://mazzo.li/epilogue/index.html%3Fp=1098.html
20. Miquel, A.: Le calcul des constructions implicites. Ph.D. thesis, Université Paris Diderot (2001). URL https://github.com/coq-contribs/paradoxes/blob/master/Russell.v
21. Paulin-Mohring, C.: Inductive definitions in the system coq rules and properties. In: M. Bezem, J.F. Groote (eds.) Typed Lambda Calculi and Applications, pp. 328–345. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
22. Pujet, L., Tabareau, N.: Observational Equality: Now For Good. Proceedings of the ACM on Programming Languages **6**(POPL), 1–29 (2022). https://doi.org/10.1145/3498693. URL https://hal.inria.fr/hal-03367052
23. Pujet, L., Tabareau, N.: Impredicative observational equality. Proc. ACM Program. Lang. **7**(POPL) (2023). https://doi.org/10.1145/3571739. URL https://doi.org/10.1145/3571739
24. Pujet, L., Tabareau, N.: A Logical Relation for Observational Equality Meets CIC (2024). https://doi.org/10.5281/zenodo.10499152. URL https://doi.org/10.5281/zenodo.10499152
25. Swan, A.: An algebraic weak factorisation system on 01-substitution sets: a constructive proof. Journal of Logic and Analysis (2016). https://doi.org/10.4115/jla.2016.8.1. URL http://dx.doi.org/10.4115/jla.2016.8.1
26. Timany, A., Sozeau, M.: Consistency of the predicative calculus of cumulative inductive constructions (pcuic) (2020)
27. Vezzosi, A., Mörtberg, A., Abel, A.: Cubical agda: A dependently typed programming language with univalence and higher inductive types. Proc. ACM Program. Lang. **3**(ICFP) (2019). https://doi.org/10.1145/3341691. URL https://doi.org/10.1145/3341691
28. Werner, B.: On the strength of proof-irrelevant type theories **4**, 1–20 (2008)

Definitional Functoriality for Dependent (Sub)Types

Théo Laurent[1], Meven Lennon-Bertrand[2], and Kenji Maillard[1](✉)

[1] Inria, Paris, France
{theo.laurent,kenji.maillard}@inria.fr
[2] University of Cambridge, Cambridge, United Kingdom
Meven.Lennon-Bertrand@cl.cam.ac.uk

Abstract. Dependently typed proof assistant rely crucially on definitional equality, which relates types and terms that are automatically identified in the underlying type theory. This paper extends type theory with definitional *functor laws*, equations satisfied propositionally by a large class of container-like type constructors F: Type → Type, equipped with a $\text{map}_F : (A \to B) \to F\,A \to F\,B$, such as lists or trees. Promoting these equations to definitional ones strengthens the theory, enabling slicker proofs and more automation for functorial type constructors. This extension is used to modularly justify a structural form of coercive subtyping, propagating subtyping through type formers in a map-like fashion. We show that the resulting notion of coercive subtyping, thanks to the extra definitional equations, is equivalent to a natural and implicit form of subsumptive subtyping. The key result of decidability of typechecking in a dependent type system with functor laws for lists has been entirely mechanized in Coq.

Keywords: Subtyping · Dependent types · Logical relation.

1 Introduction

Dependent type theory is the foundation of many proof assistants: Coq [53], Lean [43], Agda [5], Idris [14], F* [51]. At its heart lies definitional equality, an equational theory that is automatically decided by the implementation of these proof systems. The more expressive definitional equality is, the less work is requested from users to identify objects. However, there is a fundamental tension at play: making the equational theory too rich leads to both practical and theoretical issues, the most prominent one being the undecidability of definitional equality. This default plagues the otherwise appealing Extensional Type Theory (ETT) [39], a type theory which makes every provable equality definitional, thus making ETT rather impractical as a basis for a proof assistant [15]. As a result, to design usable proof assistants we need to carve out a well-behaved equational theory, that strikes the right balance between expressivity and decidability. In this paper, we show that we can maintain this subtle balance while extending intensional type theory with map operations making the functorial character

© The Author(s) 2024
S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 302–331, 2024.
https://doi.org/10.1007/978-3-031-57262-3_13

of type formers explicit, and satisfying *definitional functor laws*. We prove in particular that definitional equality and type-checking remain decidable in this extension, that we dub $\mathrm{MLTT}_{\mathrm{map}}$.

The map primitives introduced in $\mathrm{MLTT}_{\mathrm{map}}$ have a computational behaviour reminiscent of *structural subtyping*, which propagates existing subtyping structurally through type-formers, and should satisfy reflexivity and transitivity laws similar to the functor laws. Guided by the design of $\mathrm{MLTT}_{\mathrm{map}}$, we devise a second system, $\mathrm{MLTT}_{\mathrm{coe}}$, with explicit coercions witnessing structural subtyping. To gauge the expressivity of $\mathrm{MLTT}_{\mathrm{coe}}$, we relate it to a third system, $\mathrm{MLTT}_{\mathrm{sub}}$, where subtyping is implicit, as users of a type system should expect. A simple translation $|\cdot|$ from $\mathrm{MLTT}_{\mathrm{coe}}$ to $\mathrm{MLTT}_{\mathrm{sub}}$ erases coercions. We show that this erasure can be inverted, elaborating coercions back. For this to be type preserving, it is crucial that $\mathrm{MLTT}_{\mathrm{coe}}$ satisfies our new definitional equalities, which allows us to reflect the equations implicitly satisfied in $\mathrm{MLTT}_{\mathrm{sub}}$ due to coercions being transparent. Fig. 1 synthesizes the three theories that we introduce and their relationships. They all extend Martin-Löf Type Theory (MLTT) [39]. Let us now explore in more detail these three systems.

$$\begin{tikzcd}
& \mathrm{MLTT} \arrow[dl, hook] \arrow[d, hook] \arrow[dr, hook] & \\
\mathrm{MLTT}_{\mathrm{sub}} & \mathrm{MLTT}_{\mathrm{coe}} \arrow[l, two heads, "\sim", "|\cdot|"'] \arrow[r, dashed, "[\![\cdot]\!]"'] & \mathrm{MLTT}_{\mathrm{map}}
\end{tikzcd}$$

Fig. 1. Relation between MLTT, $\mathrm{MLTT}_{\mathrm{map}}$, $\mathrm{MLTT}_{\mathrm{coe}}$ and $\mathrm{MLTT}_{\mathrm{sub}}$. Arrows denote type-and-conversion-preserving translations between type theories. The dashed arrow is conjectural.

Functors and Their Laws The notion of functor is pervasive both in mathematics [38] and functional programming [34], capturing the concept of *a parametrized construction applying to objects and their transformations*. Reformulated in type theory, a type former $F\colon \mathrm{dom}(F) \to \mathrm{Type}$ is a functor when it is equipped with an operation $\mathrm{map}_F\ f\colon F\ A \to F\ B$ for any morphism $f\colon \mathrm{hom}_F(A, B)$ between two objects A,B in the domain $\mathrm{dom}(F)$ of F. Here, $\mathrm{dom}(F)$ must be endowed with the structure of a category[3], with specified composition $\circ^F$ and identities id^F, and map_F must preserve those:

$$\mathrm{map}_F\ \mathrm{id}^F = \mathrm{id} \qquad \text{(id-eq)}$$

$$(\mathrm{map}_F\ f) \circ (\mathrm{map}_F\ g) = \mathrm{map}_F\ (f \circ^F g) \qquad \text{(comp-eq)}$$

[3] Morphisms $\mathrm{hom}_F(A, B)$ in $\mathrm{dom}(F)$ are not constrained to be type theoretic functions. Accordingly, composition need not to be literally the composition of functions and the specified identities can differ from the identity $\lambda x.x$.

These two equations are known as the *functor laws.* For many container-like functors, such as **List** A, lists of elements taken in a type A, a map function can be defined in vanilla type theory such that these equations can be shown *propositionally*, *e.g.* by induction. Such propositional equations need however to be used explicitly, putting an extra burden on users and possibly causing coherence issues typical when working with propositional equalities [54]. This is not acceptable: such simple and natural identifications should hold definitionally!

Example 1 (Representation Change). Consider a dataset of pairs of a number and a boolean, represented as a list. For compatibility purpose, we may need to embed these pairs into a larger dataset using

$$\begin{gathered}\texttt{glue}\ (r\colon\ \{a\colon \mathbf{N}; b\colon \mathbf{B}\})\colon \{x\colon \mathbf{B}; y\colon \mathbf{N}; z\colon \mathbf{N}\} \overset{\text{def}}{=} \\ \{x := r.b; y := r.a; z := \texttt{if}\ r.b\ \texttt{then}\ r.a\ \texttt{else}\ 42\}.\end{gathered}$$

Going from one dataset to the other amounts to mapping either `glue` or its left inverse `glue_retr`, which forgets the extra field:

$$\begin{aligned}\text{map}_{\mathbf{List}}\ \texttt{glue} &: \mathbf{List}\ \{a\colon \mathbf{N}; b\colon \mathbf{B}\} \to \mathbf{List}\ \{x\colon \mathbf{B}; y\colon \mathbf{N}; z\colon \mathbf{N}\}, \\ \text{map}_{\mathbf{List}}\ \texttt{glue\_retr} &: \mathbf{List}\ \{x\colon \mathbf{B}; y\colon \mathbf{N}; z\colon \mathbf{N}\} \to \mathbf{List}\ \{a\colon \mathbf{N}; b\colon \mathbf{B}\}.\end{aligned}$$

If the functor laws only hold propositionally, each consecutive simplification of back and forth changes of representation needs to be explicitly lifted to lists, and applied. The uncontrolled accumulation of repetitive proof steps, even as simple as these, can quickly burden proof development. In presence of definitional functor laws, instead, any sequence of representation changes will reduce to a single $\text{map}_{\mathbf{List}}$: the boilerplate of explicitly manipulating the functor laws is handled automatically by the type theory. Moreover, observe that in this example the retraction $\texttt{glue\_retr} \circ \texttt{glue} \cong \text{id}$ is definitional thanks to surjective pairing. Combined with definitional functor laws, the following simplification step is discharged automatically by the type-checker:[4]

$$\text{map}_{\mathbf{List}}\ \texttt{glue\_retr}\ (\text{map}_{\mathbf{List}}\ \texttt{glue}\ l) \cong \text{map}_{\mathbf{List}}\ \text{id}\ \ l \cong l$$

Note that these equations are valid in any context, in particular under binders, whereas for propositional identifications, rewriting under binders is only possible in presence of the additional axiom of function extensionality.

Example 2 (Coherence of Coercions). Proof assistants may provide the ability for users to declare automatically-inserted functions acting as glue code (coercions in Coq, instance arguments in Agda, `has_coe` typeclass in Lean). Working with natural ($\mathbf{N}$), integer ($\mathbf{Z}$) and rational ($\mathbf{Q}$) numbers, we want every $\mathbf{N}$ to be automatically coerced to an integer, and so declare a `natToZ` coercion.

[4] We formalize this example, showing that this conversion indeed holds in our system, in file Example_1_1.

Similarly, we can also declare a `ZToQ` coercion. If we write 0 (a $\mathbf{N}$) where a $\mathbf{Q}$ is expected, this is accepted, and 0 is silently transformed to `ZToQ` (`natToZ` 0).

Now, if we want the same mechanism to apply when we pass the list $[0 :: 1 :: 2]$ to a function expecting a **List** $\mathbf{Q}$, we need to provide a way to propagate the coercions on lists. We can expect to solve this problem by declaring $\mathrm{map}_{\mathbf{List}}$ as a coercion, too: whenever there is a coercion $f : A \to B$, then $\mathrm{map}_{\mathbf{List}}\, f$ should be a coercion from **List** A to **List** B. However, by doing so, we would cause more trouble than we solve, as there would be two coercions from **List** $\mathbf{N}$ to **List** $\mathbf{Q}$, $\mathrm{map}_{\mathbf{List}}(\texttt{ZToQ} \circ \texttt{natToZ})$ and $(\mathrm{map}_{\mathbf{List}}\, \texttt{ZToQ}) \circ (\mathrm{map}_{\mathbf{List}}\, \texttt{natToZ})$. In the absence of definitional functor laws for $\mathrm{map}_{\mathbf{List}}$, these two are *not* definitionally equal. To add insult to injury, coercions are by default not printed to the user, yielding puzzling error messages like "*l and l are not convertible*" (!), because one is secretly $\mathrm{map}_{\mathbf{List}}(\texttt{ZToQ} \circ \texttt{natToZ})\ l$ while the other is $\mathrm{map}_{\mathbf{List}}\, \texttt{ZToQ}\ (\mathrm{map}_{\mathbf{List}}\, \texttt{natToZ}\ l)$. This makes $\mathrm{map}_{\mathbf{List}}$ virtually unusable with coercions.

Structural Subtyping This last example suggests a connection with *subtyping*. Subtyping equips the collection of types with a *subtyping order* $\preccurlyeq$ that allows to seamlessly transport terms from a subtype to a supertype, *i.e.* from A to A' when $A \preccurlyeq A'$. An important aspect of subtyping is *structural subtyping*, *i.e.* how subtyping extends structurally through type formers of the type theory. Typically, we want to have **List** $A \preccurlyeq$ **List** A' whenever $A \preccurlyeq A'$. In the context of the F* program verification platform that heavily uses refinement subtyping, the inability to propagate subtyping on inductive datatypes such as lists has been a long-standing issue that never got solved properly [25]. The absence of structural subtyping also has a history of causing difficulties to Agda [16, 22].

Definitional Equalities for Subtyping From the perspective of users of interactive theorem prover, subtyping should be implicit, transparently providing the expected glue to smoothen the writing of complex statements. From a metatheoretical perspective, on the other hand, it is useful to explicitly represent all the necessary information of a typing derivation, including where subtyping is used. The first approach is known as *subsumptive subtyping*, on the left, whereas the latter is embodied by *coercive subtyping*, on the right:

$$\textsc{Sub}\ \frac{\Gamma \vdash_{\mathrm{sub}} t : A \qquad \Gamma \vdash_{\mathrm{sub}} A \preccurlyeq A'}{\Gamma \vdash_{\mathrm{sub}} t : A'} \qquad\qquad \textsc{Coe}\ \frac{\Gamma \vdash_{\mathrm{coe}} t : A \qquad \Gamma \vdash_{\mathrm{coe}} A \preccurlyeq A'}{\Gamma \vdash_{\mathrm{coe}} \mathrm{coe}_{A,A'}\, t : A'}$$

We want to present subsumptive subtyping to users, but ground the system on the algebraic, better-behaved coercive subtyping. Informally, an application of Sub in the subsumptive type theory $\mathrm{MLTT}_{\mathrm{sub}}$ should correspond to an application of Coe in the coercive type theory $\mathrm{MLTT}_{\mathrm{coe}}$. However, given a derivation $\mathcal{D}$ of $\Gamma \vdash_{\mathrm{sub}} t : A$ we can apply Sub together with a reflexivity proof $\Gamma \vdash_{\mathrm{sub}} A \preccurlyeq A$ to yield a new derivation $\mathcal{D}'$ with the same conclusion $\Gamma \vdash_{\mathrm{sub}} t : A$. $\mathcal{D}$ and $\mathcal{D}'$ should respectively correspond to terms $\Gamma \vdash_{\mathrm{coe}} t' : A$ and $\Gamma \vdash_{\mathrm{coe}} \mathrm{coe}_{A,A}\, t' : A$ in $\mathrm{MLTT}_{\mathrm{coe}}$. Since t' and $\mathrm{coe}_{A,A}\, t'$ both erase to the same $\mathrm{MLTT}_{\mathrm{sub}}$ term t, they need to be equated if we want both type theories to be equivalent. Similarly, transitivity of subtyping implies that coercions should compose definitionally, that is $\Gamma \vdash_{\text{coe}} \text{coe}_{B,C}(\text{coe}_{A,B}\, t') \cong \text{coe}_{A,C}\, t' : C$ should always hold in MLTT_{coe}.

Functor Laws Meet Structural Subtyping Luo et al. [36] showed that the functorial composition law comp-eq is enough to make structural coercive subtyping compose definitionally, because a coercion between lists $\text{coe}_{\textbf{List}\,A,\textbf{List}\,B}$ behaves just as the function obtained by mapping $\text{coe}_{A,B}$ on every element of the list. We further investigate this bridge between coercive subtyping and functoriality of type formers, in particular the identity functor law id-eq needed to handle reflexivity of subtyping, and extend Luo et al.'s limited type system to full-blown Martin-Löf Type Theory (MLTT), with universes and large elimination. This understanding leads to a modular design of subtyping: structural subtyping for a type former relies on a functorial structure, and can be considered orthogonally to other type formers of the theory or to the base subtyping. Moreover, definitional functor laws are sufficient to make structural coercive subtyping for any type former expressive and flexible enough to interpret subsumptive subtyping.

Contributions We make the following contributions:

- we design MLTT_{map}, an extension of MLTT exhibiting the functorial nature of standard type formers ($\Pi, \Sigma, \textbf{List}, \textbf{W}, \textbf{Id}, +$), with definitional functor laws (Section 3);
- we mechanize the metatheory of a substantial fragment of MLTT_{map} in Coq, extending a formalization of MLTT [3], proving it is normalizing and has decidable type-checking (Section 4);
- we develop bidirectional presentations for MLTT_{sub} and MLTT_{coe}, which extend MLTT respectively with subsumptive and coercive subtyping;
- we leverage these presentations and the extra functorial equations satisfied by coe in MLTT_{coe} to give back and forth, type-preserving translations between the two systems (Section 5).

Detailed proofs and complete typing rules can be found in the extended version of this paper [32]. The mechanized metatheory of Section 4 is provided in [31], completed with a note describing further formalization details.

2 Type Theory and Its Metatheory

We work in the setting of dependent type theories *à la* Martin-Löf (MLTT) [39], an ideal abstraction of the type theories underlying existing proof assistants such as Agda, Coq, F* or Lean. MLTT employs five categories of judgements, characterizing the well-formed contexts ($\vdash \Gamma$), types ($\Gamma \vdash T$) and terms ($\Gamma \vdash t : T$), and providing the equational theory on types ($\Gamma \vdash A \cong B$) and terms ($\Gamma \vdash t \cong u : A$). Two terms related by this equational theory are said to be *definitionally equal* or *convertible*.

Negative Types: Dependent Products and Sums Dependent function types, noted $\Pi\, x{:}\, A.B$, are introduced using λ-abstraction $\lambda\, x{:}\, A.t$ and eliminated with application $t\, u$. We also include dependent sum types $\Sigma\, x{:}\, A.B$, introduced with pairs $(t, u)_{x.B}$ and eliminated through projections $\pi_1\, p$ and $\pi_2\, p$. Both come with an η-law.

Universes of Types Our type theories feature a countable hierarchy of universes Type_i. Any inhabitant of a universe is a well-formed type, and, in order to make the presentation compact, we do not repeat rules applying both for universes and types, implicitly assuming that a rule given for terms of some universe Type_i has a counterpart as a type judgement whenever it makes sense.

Positive Types: Inductives As we study the functorial status of type formers, *parametrized* inductive types are our main focus. Our running example is the type of lists $\mathbf{List}\, A$, parametrized by a type A, and inhabited by the empty list ε_A and the consing $hd ::_A tl$ of a head $hd : A$ onto a tail $tl : \mathbf{List}\, A$. Lists are eliminated using the dependent eliminator $\mathrm{ind}_{\mathbf{List}\, A}(s; l.P; b_\varepsilon, x.y.z.b_{::})$, which performs induction on the scrutinee s, returning a value in $P[s]$, using the two branches $b_{::}$ and b_ε. More generally, strictly positive recursive datatypes are often presented in MLTT via $\mathbf{W}\, x{:}\, A.B$, the type of well-founded trees with nodes labelled by $a : A$ of arity $B\, a$. Finally, Martin-Löf's identity type $\mathbf{Id}\; A\, x\, y$ represents equalities between two elements $x, y{:}\, A$. A general inductive type scheme is outside the scope of this paper, but the specific types we treat ($\mathbf{List}$, $\mathbf{W}$, $\mathbf{Id}$ and $+$) cover all aspects of inductive types: recursion, branching, parameters, and indices. Moreover, they can emulate all indexed inductive types [1, 7, 26], although we will see in Section 3.1 that this encoding interacts poorly with functor laws.

Rules in the Paper Due to space constraints, we focus in the text on the most interesting rules, and on two types: dependent functions and lists. Together, they cover the interesting points of our work: dependent product types have a binder and come with an η-law; lists are a parametrized datatype, for which definitional functor laws are challenging. Complete rules are given in the appendix of [32].

2.1 Metatheoretical Properties

In order to show that the extensions of MLTT from Figure 1 are well-behaved, we establish the following meta-theoretical properties.

In order to be logically sound, a type theory should have no closed term of the empty type, *i.e.* there should be no t such that $\vdash t : \mathbf{0}$. This *consistency* property is an easy consequence of *canonicity*, which characterizes the inhabitants of inductive types in the empty context as those obtained by repeated applications of constructors, up to conversion. Consistency follows, as $\mathbf{0}$ has no constructor.

A proof assistant should also be able to check whether a proof is valid, *i.e.* whether a typing judgement is derivable. In a dependent type system where terms essentially encode the structure of derivations, the main obstacle to decidability of typing is that of conversion.

In order to establish both consistency and decidability, we exhibit a function computing *normal forms* of terms. Inspecting the possible normal forms in the empty context entails canonicity. Moreover, conversion of normal forms is easily decided, and so we can build on normalization to decide conversion. Finally, we can go further, and use normalization to build canonical representatives of typing and conversion derivations, which we rely on to relate our different systems.

A more technical, but equally important property is injectivity of type constructors, for instance that whenever $\Pi\, x\colon A.B \cong \Pi\, x\colon A'.B'$, then $A \cong A'$ and $B \cong B'$. For dependent type theories, injectivity of type constructors is the main stepping stone towards subject reduction, the fact that reduction is type-preserving, and thus included in conversion.

$\boxed{t \rightsquigarrow^1 t'}$ Term t weak-head reduces in one step to term t'

$$\frac{}{(\lambda\, x\colon A.t)\ u \rightsquigarrow^1 t[u]} \qquad \frac{}{\mathrm{ind}_{\mathbf{List}\,A}(\varepsilon_A; x.P; b_\varepsilon, x.y.z.b_{::}) \rightsquigarrow^1 b_\varepsilon}$$

$$\frac{}{\mathrm{ind}_{\mathbf{List}\,A}(a ::_A l; x.P; b_\varepsilon, x.y.z.b_{::}) \rightsquigarrow^1 b_{::}[a, l, \mathrm{ind}_{\mathbf{List}\,A}(l; x.P; b_\varepsilon, x.y.z.b_{::})]}$$

$$\frac{t \rightsquigarrow^1 t'}{t\ u \rightsquigarrow^1 t'\ u} \qquad \frac{t \rightsquigarrow^1 t'}{\mathrm{ind}_{\mathbf{List}\,A}(t; x.P; b_\varepsilon, x.y.z.b_{::}) \rightsquigarrow^1 \mathrm{ind}_{\mathbf{List}\,A}(t'; x.P; b_\varepsilon, x.y.z.b_{::})}$$

$\boxed{t \rightsquigarrow^\star t'}$ Term t weak-head reduces in multiple steps to term t'

$$\frac{}{t \rightsquigarrow^\star t} \qquad \frac{t \rightsquigarrow^1 t' \qquad t' \rightsquigarrow^\star t''}{t \rightsquigarrow^\star t''}$$

$\boxed{\mathrm{nf}\quad f} \overset{\text{def}}{=} n \mid \Pi\, x\colon t.t \mid \mathrm{Type}_i \mid \mathbf{List}\, t \mid \lambda\, x\colon A.t \mid \varepsilon_A \mid t ::_A t$ weak-head normal forms

$\boxed{\mathrm{ne}\quad n} \overset{\text{def}}{=} x \mid n\ t \mid \mathrm{ind}_{\mathbf{List}\,A}(n; t; t, t)$ weak-head neutrals

Fig. 2. Weak-head reduction and normal forms (t stands for an arbitrary term)

2.2 Neutrals, Normals, and Reduction

Before getting to how we establish these properties, we must introduce a last element: computation. Indeed, most conversion rules can be seen not just as equalities but be oriented as computations to be performed. This leads to the definition of weak-head reduction $\rightsquigarrow^\star$ in Figure 2. Weak-head reduction is the only reduction that is used throughout this article.

The normal forms (nf) for weak-head reduction, *i.e.* the terms that cannot reduce, are inductively characterized at the bottom of Figure 2, together with the companion notion of neutral forms (ne). Normal forms can be either a canonical term, starting with a head constructor (for instance, a λ-abstraction or ε), or a neutral term. Neutrals are stuck computations, blocked by a variable, *e.g.* $x\ u$ is stuck on x and cannot reduce further.

2.3 Proof techniques

Logical Relations Logical relations are our main tool to obtain normalization and canonicity results. At a high-level, we follow the approach of Abel et al. [2], where the logical relation is based on reducibility, a complex predicate on types and terms, which in particular entails the existence of a weak-head normal form. The key property is the fundamental lemma, stating that every well-typed term is reducible, *i.e.* that the logical relation is a model of MLTT. The existence of (deep) normal forms is obtained through the inspection of reducibility derivations for a term, since they contain iterated reduction steps to a normal form.

We use the logical relation not only to characterize the normal forms of terms but also the conversion between them, showing that a proof of convertibility between two terms can be transformed to a canonical shape interleaving weak-head reduction sequences and congruence steps between weak-head normal forms. We detail in Section 4 the novel challenges we encountered when adapting the approach of Abel et al. [2] to parametrized inductive types.

Bidirectional Typing and Algorithmic Conversion Our second tool is a presentation of conversion and typing that, while still inductively defined, is as close as possible to an actual implementation. Typing is bidirectional [30, 44], *i.e.* decomposed into type inference and type checking, and essentially follows Lennon-Bertrand [30].[5] We use bidirectional typing for its rigid, canonical derivation structure, rather than for its ability to cut down type annotations on terms. Thus, although we use bidirectional judgements, all our terms infer a type, in contrast to what is common in the bidirectional literature [20, 41].

Algorithmic conversion, presented in Figure 3 combines ideas from both bidirectional typing and the presentation of Abel et al. [2]. Crucially, it gets rid entirely of the generic transitivity rule for conversion, and instead uses term-directed reduction, intertwined with comparison of the heads of weak-head normal forms. Algorithmic conversion is mutually defined with a second relation, dedicated to comparing weak-head neutral forms, called when encountering neutrals at positive types. General conversion is "checking", *i.e.* taking a type as input, while neutral comparison is "inferring", *i.e.* the type is an output. In turn, conversion is used in the following typing rule to compare the inferred type for t with the one it should check against.

$$\textsc{Check}\ \frac{\Gamma \vdash t \rhd T' \qquad \Gamma \vdash T' \cong T \lhd}{\Gamma \vdash t \lhd T}$$

Using the consequences of the logical relation, we can show that this algorithmic presentation has many desirable properties. For instance, transitivity

[5] In line with Lennon-Bertrand [30], we pick $\rhd$ as the symbol for inference, and $\lhd$ as the one for checking, to avoid clashes with Coq's `=>` in the formalization.

$\boxed{\Gamma \vdash n \approx n' \rhd T}$ Neutrals n and n' are comparable, inferring the type T

$$\textsc{NVar}\ \frac{(x\colon T) \in \Gamma}{\Gamma \vdash x \approx x \rhd T} \qquad \textsc{NApp}\ \frac{\Gamma \vdash n \approx_{\mathrm{h}} n' \rhd \Pi\, x\colon A.B \qquad \Gamma \vdash u \cong u' \lhd A}{\Gamma \vdash n\ u \approx n'\ u' \rhd B[u]}$$

$\boxed{\Gamma \vdash t \cong_{\mathrm{h}} t' \lhd T}$ Reduced terms t and t' are convertible at type T

$$\textsc{CList}\ \frac{\Gamma \vdash A \cong A' \lhd \mathrm{Type}_i}{\Gamma \vdash \mathbf{List}\, A \cong_{\mathrm{h}} \mathbf{List}\, A' \lhd \mathrm{Type}_i} \qquad \textsc{CProd}\ \frac{\Gamma \vdash A \cong A' \lhd \mathrm{Type}_i \quad \Gamma, x\colon A' \vdash B \cong B' \lhd \mathrm{Type}_i}{\Gamma \vdash \Pi\, x\colon A.B \cong_{\mathrm{h}} \Pi\, x\colon A'.B' \lhd \mathrm{Type}_i}$$

$$\textsc{CFun}\ \frac{\Gamma, x\colon A \vdash f\ x \cong f'\ x \lhd B}{\Gamma \vdash f \cong_{\mathrm{h}} f' \lhd \Pi\, x\colon A.B} \qquad \textsc{CCons}\ \frac{\Gamma \vdash a \cong a' \lhd A'' \qquad \Gamma \vdash l \cong l' \lhd \mathbf{List}\, A''}{\Gamma \vdash a ::_A l \cong_{\mathrm{h}} a' ::_{A'} l' \lhd \mathbf{List}\, A''}$$

$$\textsc{NeuList}\ \frac{\Gamma \vdash n \approx_{\mathrm{h}} n' \rhd S}{\Gamma \vdash n \cong_{\mathrm{h}} n' \lhd \mathbf{List}\, A} \qquad \textsc{NeuNeu}\ \frac{\mathrm{ne}\, M \quad \Gamma \vdash n \approx n' \rhd N}{\Gamma \vdash n \cong_{\mathrm{h}} n' \lhd M}$$

$\boxed{\Gamma \vdash t \cong t' \lhd T}$ Terms t and t' are convertible at type T

$\boxed{\Gamma \vdash n \approx_{\mathrm{h}} n' \rhd T}$ Neutrals n and n' are comparable, inferring reduced type T

$$\textsc{TmRed}\ \frac{t \leadsto^\star u \qquad t' \leadsto^\star u' \quad T \leadsto^\star U \qquad \Gamma \vdash u \cong_{\mathrm{h}} u' \lhd U}{\Gamma \vdash t \cong t' \lhd T} \qquad \textsc{NRed}\ \frac{\Gamma \vdash n \approx n' \rhd T \quad T \leadsto^\star S \qquad \mathrm{nf}\, S}{\Gamma \vdash n \approx_{\mathrm{h}} n' \rhd S}$$

Fig. 3. Algorithmic conversion

is admissible, even though there is no dedicated rule. Collecting the properties derived from the logical relation, we can obtain our second main objective: equivalence between the algorithmic and declarative presentations.

Property 1 (Equivalence of the Presentations). If $\Gamma \vdash t : T$, then $\Gamma \vdash t \lhd T$. Conversely, if $\vdash \Gamma$, $\Gamma \vdash T$ and $\Gamma \vdash t \lhd T$, then $\Gamma \vdash t : T$.

Note that the implication from the bidirectional judgement to the declarative one only holds if the context and type are well-formed. In general, our algorithmic presentations are "garbage-in, garbage-out": they maintain well-formation of types and contexts, but do not enforce them. Thus, most properties of the algorithmic derivations only hold if their inputs are well-formed, in the sense of Figure 4. Note that in checking and inference modes, while the term is an input, it is of course not assumed to be well-formed in advance, since this is what the judgement itself asserts. This algorithmic, syntax-directed presentation is well suited for implementations and to establish relationships between type systems.

Judgement	Input(s)	Inputs are well-formed
$\Gamma \vdash t \rhd T$	Γ, t	$\vdash \Gamma$
$\Gamma \vdash t \lhd T$	Γ, T, t	$\vdash \Gamma$ and $\Gamma \vdash T$
$\Gamma \vdash T \cong T' \lhd$	Γ, T and T'	$\vdash \Gamma$, $\Gamma \vdash T$ and $\Gamma \vdash T'$
$\Gamma \vdash t \cong t' \lhd T$	Γ, t, t' and T	$\vdash \Gamma$, $\Gamma \vdash T$, $\Gamma \vdash t : T$ and $\Gamma \vdash t' : T$
$\Gamma \vdash t \approx t' \rhd T$	Γ, t and t'	$\vdash \Gamma$, $\mathrm{ne}\, t$, $\mathrm{ne}\, t'$, and $\exists A, A'$ s.t. $\Gamma \vdash t : A$, $\Gamma \vdash t' : A'$

Fig. 4. Well-formed inputs (for $\cong_{\mathrm{h}}, \approx_{\mathrm{h}}, \rhd_{\mathrm{h}}$, similar to their non-reduced variants)

3 A Functorial Type Theory

We develop an extension $\mathrm{MLTT}_{\mathrm{map}}$ of MLTT with primitive map_F operations for each parametrized type former F of MLTT, that is Π, Σ,$+$, **List**, **W**, and **Id**. These map operations internalize the functorial character of the type formers,[6] and by design *definitionally* satisfy the functor laws for each type former F:

$$\mathrm{map}_F\ \mathrm{id} \cong \mathrm{id} \qquad \text{(id-eq)}$$

$$\mathrm{map}_F\ f \circ \mathrm{map}_F\ g \cong \mathrm{map}_F\ (f \circ g) \qquad \text{(comp-eq)}$$

Section 3.1 describes the structure needed on type formers to state their functoriality in $\mathrm{MLTT}_{\mathrm{map}}$. In Section 3.2 we show how definitionally functorial map_F are definable in vanilla MLTT for type formers with an η-law. Section 3.3 introduces the main content of this paper, required to enforce the functor laws on inductive type formers: the extension of the equational theory on neutral terms. We explain the technical design choices needed to define and use the logical relations for $\mathrm{MLTT}_{\mathrm{map}}$ and obtain as a consequence that the theory enjoys consistency, canonicity, and decidable conversion and type-checking. We implement these design choices in Coq for a simplified but representative version of $\mathrm{MLTT}_{\mathrm{map}}$, with one universe and the Π, Σ, **List** and **N** type formers, with their respective map operators. This formalization is detailed in Section 4.

3.1 Functorial Structure on Type Formers

In order to state the functor laws for a type former F, such as $\Pi, \Sigma,$ **List**, **W**, **Id**, we must specify the categorical structures involved. A type former F is parametrized by a telescope of parameters that we collectively refer to as $\mathrm{dom}(F)$, and produces a type. We will always equip the codomain Type of a type former F with the category structure of functions between types, with the standard identity and composition. Note that composition is associative and unital up to conversion, thanks to η-laws on function types.

The domain $\mathrm{dom}(F)$ of a type former must also be equipped with the structure of a category. We introduce the judgement $\Delta \vdash_{\mathrm{map}} X : \mathrm{dom}(F)$ to stand for

[6] These equations are all propositionally true in MLTT, proven by induction for datatypes.

a substitution in context Δ of the telescope of parameters of F. Then, given two such instances X_1 and X_2 of parameters for F, morphisms between X_1 and X_2 are classified by the judgement $\Delta \vdash_{\mathrm{map}} \varphi : \mathrm{hom}_F(X_1, X_2)$. We require $\mathrm{dom}(F)$ to be also equipped with identities and a definitionally associative and unital composition:

$$\frac{\Delta \vdash_{\mathrm{map}} X : \mathrm{dom}(F)}{\Delta \vdash_{\mathrm{map}} \mathrm{id}^F_X : \mathrm{hom}_F(X, X)} \qquad \frac{\Delta \vdash_{\mathrm{map}} \varphi : \mathrm{hom}_F(X, Y) \qquad \Delta \vdash_{\mathrm{map}} \psi : \mathrm{hom}_F(Y, Z)}{\Delta \vdash_{\mathrm{map}} \psi \circ^F \varphi : \mathrm{hom}_F(X, Z)}$$

For instance, for dependent products, $\mathrm{dom}(\Pi)$ and hom_Π are given by

$$\begin{aligned} \Delta \vdash_{\mathrm{map}} (A, B) : \mathrm{dom}(\Pi) &\iff \Delta \vdash_{\mathrm{map}} A \wedge \Delta, a \colon A \vdash_{\mathrm{map}} B \\ \Delta \vdash_{\mathrm{map}} (f, g) : \mathrm{hom}_\Pi((A_1, B_1), (A_2, B_2)) &\iff \Delta \vdash_{\mathrm{map}} f : A_2 \to A_1 \quad \wedge \\ &\qquad \Delta, a \colon A_2 \vdash_{\mathrm{map}} g : B_1[f\, a] \to B_2 \end{aligned}$$

with identity $\mathrm{id}^\Pi_{(A,B)} \overset{\mathrm{def}}{=} (\mathrm{id}_A, \mathrm{id}_B)$ and composition $(f, g) \circ^\Pi (f', g') \overset{\mathrm{def}}{=} (f' \circ f, g \circ g')$.

The domain and morphism for each type former are described in Figure 5. Identities and compositions are given by the categorical structure on Type for **List** and **Id**, and are defined componentwise, for Σ, **W** and $+$, similarly to Π. Figure 6 presents the conversion rules of $\mathrm{MLTT}_{\mathrm{map}}$, extending those of MLTT

Type former F	Domain $\Delta \vdash_{\mathrm{map}} X : \mathrm{dom}(F)$	Morphisms $\Delta \vdash_{\mathrm{map}} \varphi : \mathrm{hom}_F(\cdot_1, \cdot_2)$
List	$X = (A) \wedge \Delta \vdash_{\mathrm{map}} A$	$\varphi = (f) \wedge \Delta \vdash_{\mathrm{map}} f : A_1 \to A_2$
Π	$X = (A, B) \wedge \Delta \vdash_{\mathrm{map}} A$ $\wedge \quad \Delta, a \colon A \vdash_{\mathrm{map}} B$	$\varphi = (f, g) \wedge \Delta \vdash_{\mathrm{map}} f : A_2 \to A_1$ $\wedge\, \Delta, a \colon A_2 \vdash_{\mathrm{map}} g : B_1[f\, a] \to B_2$
Σ	idem	$\varphi = (f, g) \wedge \Delta \vdash_{\mathrm{map}} f : A_1 \to A_2$ $\wedge\, \Delta, a \colon A_1 \vdash_{\mathrm{map}} g : B_1 \to B_2[f\, a]$
W	idem	$\varphi = (f, g) \wedge \Delta \vdash_{\mathrm{map}} f : A_1 \to A_2$ $\wedge\, \Delta, a \colon A_1 \vdash_{\mathrm{map}} g : B_2[f\, a] \to B_1$
Id	$X = (A, x, y) \wedge \Delta \vdash_{\mathrm{map}} A$ $\wedge \quad \Delta \vdash_{\mathrm{map}} x : A$ $\wedge \quad \Delta \vdash_{\mathrm{map}} y : A$	$\varphi = (f) \wedge \Delta \vdash_{\mathrm{map}} f : A_1 \to A_2$ $\wedge \quad \Delta \vdash_{\mathrm{map}} f\, x_1 \cong x_2 : A_2$ $\wedge \quad \Delta \vdash_{\mathrm{map}} f\, y_1 \cong y_2 : A_2$
$+$	$X = (A, B) \wedge \Delta \vdash_{\mathrm{map}} A$ $\wedge \quad \Delta \vdash_{\mathrm{map}} B$	$\varphi = (f, g) \wedge \Delta \vdash_{\mathrm{map}} f : A_1 \to A_2$ $\wedge \quad \Delta \vdash_{\mathrm{map}} g : B_1 \to B_2$

Fig. 5. Domain and categorical structure on type formers

with general functoriality rules and specific rules for each type former. For each type former F, map_F is introduced using Map and witnesses the functorial nature of F, that is F maps morphisms φ in its domain between two instances of its parameters X, Y (left implicit) to functions between types

$$\Delta \vdash_{\mathrm{map}} \varphi : \mathrm{hom}_F(X, Y) \qquad \Longrightarrow \qquad \Delta \vdash_{\mathrm{map}} \mathrm{map}_F\ \varphi : F\, X \to F\, Y$$

These mapping operations obey the two functor laws, as stated by MapId and MapComp.

The computational behaviour of maps, as defined by weak-head reduction, depends on the type former. On Π and Σ, map is defined by its observation, namely application for Π and first and second projections for Σ. On inductive types such as **List**, **W**, **Id** and $+$, map traverses constructors, applying the provided morphism on elements of the parameter type(s), and itself to recursive arguments. This corresponds to the usual notion of map on lists. On **W**-types, the map operation relabels the nodes of the trees using its first component, and reorganizes the subtrees according to its second component. On identity types, the reflexivity proof $\mathrm{refl}_{A_1,a}$ at a point $a : A_1$ is mapped to the reflexivity proof at $f\,a : A_2$ for $f : A_1 \to A_2$. On sum types $A + B$, either the first or second component of the morphism (f, g) is employed depending on the constructor inj^l or inj^r . Each reduction rule has a corresponding conversion rule in $\mathrm{MLTT}_{\mathrm{map}}$.

$\boxed{\text{For each type former } F \ (\Pi,\ \Sigma,\ \textbf{List},\ \textbf{W},\ \textbf{Id},\ +)}$

$$\textsc{Map}\ \frac{\Gamma \vdash_{\mathrm{map}} X, Y : \mathrm{dom}(F) \qquad \Gamma \vdash_{\mathrm{map}} f : \mathrm{hom}_F(X, Y)}{\Gamma \vdash_{\mathrm{map}} \mathrm{map}_F\, f : F\,X \to F\,Y} \qquad \textsc{MapId}\ \frac{\Gamma \vdash_{\mathrm{map}} X : \mathrm{dom}(F) \qquad \Gamma \vdash_{\mathrm{map}} t : F\,X}{\Gamma \vdash_{\mathrm{map}} \mathrm{map}_F\, \mathrm{id}^F_X\ t \cong t : F\,X}$$

$$\textsc{MapComp}\ \frac{\Gamma \vdash_{\mathrm{map}} X, Y, Z : \mathrm{dom}(F) \qquad \Gamma \vdash_{\mathrm{map}} g : \mathrm{hom}_F(X, Y) \qquad \Gamma \vdash_{\mathrm{map}} f : \mathrm{hom}_F(Y, Z) \qquad \Gamma \vdash_{\mathrm{map}} t : F\,X}{\Gamma \vdash_{\mathrm{map}} \mathrm{map}_F\ f\ (\mathrm{map}_F\, g\, t) \cong \mathrm{map}_F (f \circ^F g)\, t : F\,Z}$$

$\boxed{\text{Specific rules}}$

$$\mathrm{map}_{\textbf{List}}\ f\ (hd :: tl) \rightsquigarrow^1 f\,hd :: \mathrm{map}_{\textbf{List}}\ f\,tl \qquad \mathrm{map}_{\textbf{List}}\ f\,\varepsilon \rightsquigarrow^1 \varepsilon$$

$$\pi_1\,(\mathrm{map}_\Sigma\ f\,p) \rightsquigarrow^1 (\pi_1\,f)\,(\pi_1\,p) \qquad \pi_2\,(\mathrm{map}_\Sigma\ f\,p) \rightsquigarrow^1 (\pi_2\,f)\,(\pi_2\,p)$$

$$\mathrm{map}_\Pi\ f\,h\,t \rightsquigarrow^1 (\pi_2\,f)\,(h\,(\pi_1\,f\,t)) \qquad \mathrm{map}_{\textbf{Id}}\ f\ \mathrm{refl}_{A_1,a} \rightsquigarrow^1 \mathrm{refl}_{A_2,f\,a}$$

$$\mathrm{map}_+\ (f,g)\,(\mathrm{inj}^l\ a) \rightsquigarrow^1 \mathrm{inj}^l\ (f\,a) \qquad \mathrm{map}_+\ (f,g)\,(\mathrm{inj}^r\ b) \rightsquigarrow^1 \mathrm{inj}^r\ (g\,b)$$

$$\mathrm{map}_{\textbf{W}}\{T_1\}\{T_2\} f\,(\mathrm{sup}\,a\ k) \rightsquigarrow^1$$
$$\mathrm{sup}_{x.\pi_2\,T_2}(\pi_1\,f\,a)\ (\lambda\,x : (\pi_2\,T_2\,(\pi_1\,f\,a)).\ \mathrm{map}_{\textbf{W}}\ f\,(k\,(\pi_2\,g\,x)))$$

$$\textsc{RedMapComp}\ \frac{\mathrm{ne}\ n \qquad F \in \{\textbf{List}, \textbf{Id}, +, \textbf{W}\}}{\mathrm{map}_F\, f\,(\mathrm{map}_F\, g\,n) \rightsquigarrow^1 \mathrm{map}_F(f \circ^F g)\,n}$$

Fig. 6. $\mathrm{MLTT}_{\mathrm{map}}$ (extends Figures 2 and 3)

Functorial Maps and Type Former Encodings Positive sum types $A + B$ can be simulated in MLTT by the type $\Sigma\, b : \mathbf{B}\,.\delta(b, A, B)$, using the branching operation

$\delta(b, A, B) \stackrel{\text{def}}{=} \text{ind}_{\mathbf{B}}(b; z.\,\text{Type}_i; A, B)$. This encoding admits the adequate introduction and elimination rules. It induces a mapping from $\text{dom}(+)$ to $\text{dom}(\Sigma)$, sending a morphism $\Delta \vdash_{\text{map}} (f, g) : \text{hom}_{+}((A_1, B_1), (A_2, B_2))$ to the morphism $\Delta \vdash_{\text{map}} (\text{id}_{\mathbf{B}}, f \oplus g) : \text{hom}_{\Sigma}((\mathbf{B}, \delta(b, A, B)), (\mathbf{B}, \delta(b, A', B')))$ where $f \oplus g$ is

$$\Delta, b\colon \mathbf{B} \vdash_{\text{map}} \text{ind}_{\mathbf{B}}(b; z.\delta(z, A, B) \to \delta(z, A', B'); f, g) : \delta(b, A, B) \to \delta(b, A', B').$$

We can show by case analysis on $\mathbf{B}$ that this mapping satisfies the propositional functor laws. However, it falls short from satisfying the definitional ones.[7] It is thus not enough to compose map_{Σ} with this mapping to obtain a functorial action on sum types $A + B$, and explains why we add $+$ primitively.

This obstruction to inductive encodings would motivate a general definition of functorial map for a scheme of indexed inductive types. However, it seems already non-trivial to specify the categorical structure on the domain of an arbitrary inductive type, let alone generate the type and equations for the corresponding map operation. Thus, we rather concentrate on understanding the theory on quintessential examples, leaving out a general treatment to future work.

3.2 Extensional Types and Map

A type A is extensional when its elements are characterized by their observation, *i.e.* any element is convertible to its η-expansion, an elimination followed by an introduction – an equation usually called η-law. For extensional type formers, it is possible to define a map operation satisfying the functor laws. In MLTT and MLTT_{map}, both (strong) dependent sums Σ and dependent products Π have such extensionality laws, and so their map operations are definable.

$$\begin{aligned} \text{map}_{\Pi}\ \ ((g, f)\colon \text{hom}_{\Pi}((A, B), (A', B')))\ (h\colon \Pi(x\colon A)B) &\stackrel{\text{def}}{=} \lambda\, x\colon A'.f\ (h\ (g\ x)) \\ \text{map}_{\Sigma}\ \ ((g, f)\colon \text{hom}_{\Sigma}((A, B), (A', B')))\ (p\colon \Sigma(x\colon A)B) &\stackrel{\text{def}}{=} (g\ (\pi_1\, p), f\ (\pi_2\, p)) \end{aligned}$$

Lemma 1. map_{Π} *and* map_{Σ} *satisfy the definitional functor laws* MapId *and* MapComp.

The proof is immediate by unfolding the definitions of map_{Π}, map_{Σ}, applications of β-reduction and the η-rules for the preservation of identity. The accompanying artifact also shows that the functor laws hold for Coq's Π and Σ types.[8] The specific rules of Figure 6 hold by β-reduction.

3.3 New Equations for Neutral Terms in Dependent Type Theory

Inductive types in MLTT do not satisfy a definitional η-law. For identity types, the η-law is equivalent to the equality reflection principle of extensional MLTT, whose equational theory is undecidable [15, 27]. Extensionality principles for inductive types with recursive occurrences as **List** or **W** are also likely to break the

[7] This would amount to an instance of the η-law for $\mathbf{B}$.

[8] In file mapPiSigmaFunctorLaws.

decidability of the equational theory, by adapting an argument for streams [40]. The result of the previous section hence does not apply, and it is instructive to look at the actual obstruction. Consider the case of **List**, and the equation for preservation of identities:

$$\Gamma \vdash_{\text{map}} \text{map}_{\mathbf{List}}\ \text{id}_A^{\mathbf{List}}\ l \cong l : \mathbf{List}\, A. \tag{$\star$}$$

If we were to define $\text{map}_{\mathbf{List}}$ by induction on lists as is standard, we would get

$$\text{map}_{\mathbf{List}} (f\colon A \to B)\, (l\colon \mathbf{List}\ A) \overset{\text{def}}{=} \text{ind}_{\mathbf{List}}(\mathbf{List}\ B; l; \varepsilon_B, hd.tl.ih_{tl}.(f\, hd) ::_B ih_{tl})$$

We can observe that Eq. ($\star$) is validated on closed canonical terms of type **List**:

$$\begin{aligned} \text{map}_{\mathbf{List}}\, \text{id}_A\ \varepsilon_A \cong \varepsilon_A\, \text{map}_{\mathbf{List}}\, \text{id}_A\, (hd ::_A tl) \\ \cong (\text{id}_A\ hd) ::_A \text{map}_{\mathbf{List}}\, \text{id}_A\ tl \overset{\text{ind. hyp.}}{\cong} hd ::_A tl \end{aligned}$$

However, on neutral terms, typically variables, we are stuck as long as we stay within the equational theory of MLTT:

$$A\colon \text{Type}, x\colon \mathbf{List}\ A \nvdash \text{map}_{\mathbf{List}}\ \text{id}_A\ x \cong x : \mathbf{List}\, A.$$

In order to validate Eq. ($\star$), MLTT_{map} must thus at the very least extend the equational theory on neutral terms. Allais et al. [6] show in the simply-typed case that these equations between neutral terms are actually the only obstruction to functor laws, and in the remainder of this section we discuss how to adapt MLTT to this idea.

Map Composition and Compacted Neutrals The first step in order to validate the functor laws is to get as close as possible to a canonical representation during reduction. In order to deal with composition of maps, we extend reduction with RedMapComp, merging consecutive stuck maps. In order to preserve the deterministic nature of weak-head reduction, map compaction should only apply when no other rule does. To achieve this, the type former F should not be extensional, because map_Π is already handled through the η-expansion of CFun, and similarly for map_Σ. Moreover, the mapped term should be neither a canonical form where map already has a computational behaviour, nor a map itself that could fire the same rule. To control this, we separate neutrals, which cannot contain a map as their head, and *compacted neutrals*, which can start with at most one map, as shown in Figure 7 alongside normal forms. Allais et al. [6] also features a similar decomposition of normal forms into three different classes, although their normal forms for lists are more complex than ours as they validate more definitional equations than functor laws.

Map on Identities For identities, using a similar reduction-based approach is difficult: turning the equation $\Gamma \vdash_{\text{map}} \text{map}_{\mathbf{List}}\ \text{id}_A\ l \cong l : \mathbf{List}\ A$ into a reduction raises issues similar to those encountered with η-laws. Orienting it as an

$$\begin{array}{lll}
\boxed{\mathrm{nf}\quad f} & \overset{\text{def}}{=} \cdots \mid c & \text{weak-head normal forms} \\
\boxed{\mathrm{ne}\quad n} & \overset{\text{def}}{=} \cdots \mid \mathrm{ind}_{\mathbf{List}\,A}(c; t; t) & \text{weak-head neutrals} \\
\boxed{\mathrm{cne}\quad c} & \overset{\text{def}}{=} n \mid \mathrm{map}_{\mathbf{List}}\, f\, n & \text{compacted neutrals}
\end{array}$$

Fig. 7. Weak-head normal and neutrals for $\mathrm{MLTT}_{\mathrm{map}}$ (extends Figure 2)

expansion $l \rightsquigarrow^\star \mathrm{map}_{\mathbf{List}}\ \mathrm{id}_A\ l$ requires knowledge of the type to ensure the expansion only applies to lists, and is potentially non-terminating. Accommodating type-directed reduction would require a deep reworking of our setting.

As a result, just like for η on functions in rule CFun, we implement this rule as part of conversion, rather than as a reduction. We also incorporate it carefully in the notion of reducible conversion in the logical relation, where we do have access to enough properties of the type theories. Since the equation is always validated by canonical forms, we only need to enforce it on compacted neutrals. The logical relation for an inductive type I (**List**, **W**, **Id**, $+$) thus specifies that a neutral n is reducibly convertible to a compacted neutral $\mathrm{map}_I\ f\, m$, whenever the neutrals n and m are convertible and f agrees with the identity of $\mathrm{dom}(I)$ on any neutral term. See MapNeConvRedL in the next section for the exact rule.

4 Formalizing New Equations for Neutral Lists

In this section we expose the main components of the accompanying Coq formalization, which covers normalization, equivalence of declarative and algorithmic typing, decidability of type-checking, and canonicity for a subset of $\mathrm{MLTT}_{\mathrm{map}}$ with $\mathbf{0}, \mathbf{N}, \Pi, \Sigma, \mathbf{List}$ and a single universe. The formalization extends a port to Coq [3] of a previous Agda formalization [2], which has already been extended multiple times [23, 45, 46]. We focus on the challenges to establish the functor laws on lists, and direct the reader either to the Coq code, or to Abel et al. and Adjedj et al. for other details. The formalization spans ~26k lines of code, approximately 9k of which are specific to our extension with lists and definitionally functorial maps and are new compared to Adjedj et al. Text in blue refer to files in the companion artifact.

4.1 A Logical Relation with Functor Laws on List

The Coq development defines both declarative and algorithmic presentations of $\mathrm{MLTT}_{\mathrm{map}}$ and proves their equivalence through a logical relation parametrized by a generic typing interface[9] instantiated by both presentations. Beyond generic variants of the typing and conversion judgement, the interface uses two extra judgements: $\Gamma \vdash_{\mathrm{map}} t \rightsquigarrow^\star t' : A$ stating that t reduces to t' and that they are

[9] Defined in GenericTyping

both well typed at type A in context Γ; and $\Gamma \vdash_{\text{map}} n \approx n' : A$ stating that n and n' are convertible neutral terms.

Definition of the Logical Relation In presence of dependent types, the standard strategy of reducibility proofs defining reducibility of terms by induction on their types fails. Rather, reducibility of types and of terms are define mutually mutually, the latter defined out of a witness of the former, and the former reusing the latter for the universe. Following Abel et al. [2], we thus first define for each type former F what it means to be a type reducible as F, and then what it means to be a reducible term and reducibly convertible terms at such a type reducible as F. A type is then reducible if it is reducible as F for some type former F. As we extend the logical relation to handle **List** and $\text{map}_{\mathbf{List}}$, we focus on a high level description of the reducibility of types as lists and the reducible convertibility of terms of type **List**, the most challenging elements in the definition.[10] Two points required specific attention with respect to prior work. First, to handle the fact that constructors contain their parameters, we need to impose reducible conversions between these and the parameters coming from the type. Second, in order to validate composition of map on neutrals that may contain a map, we need to equip neutrals with additional reducibility data, rather than pure typing information.

A type X is reducible as a list in context Γ, written $\Gamma \Vdash_{\mathbf{List}} X$, if it weak-head reduces to $\mathbf{List}\, A$ for some parameter type A reducible in any context Δ extending Γ via a weakening $\rho : \texttt{Wk}(\Delta, \Gamma)$. If $\mathfrak{R} : \Gamma \Vdash_{\mathbf{List}} X$ is a witness that X is reducible as a list, then $\mathbb{P}(\mathfrak{R})$ stands for the parameter type A of this witness, and $\mathbb{P}_{\Vdash}(\mathfrak{R}) : \Pi\{\rho : \texttt{Wk}(\Delta, \Gamma)\}.\Delta \Vdash \mathbb{P}(\mathfrak{R})[\rho]$ is its witness of reducibility.

Reducible conversion of terms as lists $\Gamma \Vdash t \cong t' : A \mid \mathfrak{R}$ is defined in Figure 8. Two terms t and t' are reducibly convertible as lists with respect to the witness of reducibility $\mathfrak{R} : \Gamma \Vdash_{\mathbf{List}} X$ if they reduce to normal forms v, v' that are reducibly convertible as normal forms of type list $\Gamma \Vdash_{\text{nf}} v \cong v' : A \mid \mathfrak{R}$ (ListRed). Straightforwardly, two canonical forms are convertible if they are both ε (NilRed) or both $- :: -$ (ConsRed) with reducibly convertible heads and tails.

For compacted neutral forms, we need to consider four cases according to whether each of the left or the right hand-side term is a $\text{map}_{\mathbf{List}}$. NeRed provides the easy case where both terms are actually neutral, with a single premise requiring that these are convertible as neutrals for the generic typing interface. MapMapConvRed gives the congruence rule for stuck $\text{map}_{\mathbf{List}}$, relating $\text{map}_{\mathbf{List}}\, f\, n$ and $\text{map}_{\mathbf{List}}\, f'\, n'$ when the mapped lists n and n' are convertible as neutrals and the bodies $f\, x$ and $f'\, x$ of the functions are reducibly convertible. Note that at this point of the logical relation, we do not know that the domain of the functions f and f' is reducible, only that their codomain is, as provided by $\mathbb{P}_{\Vdash}(\mathfrak{R})$. This constraint motivates both the η-expansion of the functions on the fly before comparing them, and the necessity of a Kripke-style quantification on larger contexts for the reducibility of the parameter type $\mathbb{P}_{\Vdash}(\mathfrak{R})$, together ensur-

[10] Available in file LogicalRelation.

ing that the recursive reducible conversion happens at a reducible type, namely an adequate instance of $\mathbb{P}(\mathfrak{R})$. Finally, the symmetric rules NeMapConvRedR and MapNeConvRedL deal with the comparison of a $\mathrm{map}_{\mathbf{List}}$ against a neutral n, that can be morally thought as $\mathrm{map}_{\mathbf{List}}$ id n, and indeed the premises correspond to what one would obtain with MapMapConvRed in that case, up to an inlined β-reduction step.

$$\textsc{ListRed}\ \frac{\Gamma \vdash_{\mathrm{map}} t \rightsquigarrow^{\star} v : \mathbf{List}\,\mathbb{P}(\mathfrak{R}) \qquad \Gamma \vdash_{\mathrm{map}} t' \rightsquigarrow^{\star} v' : \mathbf{List}\,\mathbb{P}(\mathfrak{R}) \\ \Gamma \Vdash_{\mathrm{nf}} v \cong v' : X \mid \mathfrak{R}}{\Gamma \Vdash t \cong t' : X \mid \mathfrak{R}}$$

$$\textsc{ConsRed}\ \frac{\Gamma \Vdash \mathbb{P}(\mathfrak{R}) \cong P \mid \mathbb{P}_{\Vdash}(\mathfrak{R}) \qquad \Gamma \Vdash \mathbb{P}(\mathfrak{R}) \cong P' \mid \mathbb{P}_{\Vdash}(\mathfrak{R}) \\ \Gamma \Vdash hd \cong hd' : \mathbb{P}(\mathfrak{R}) \mid \mathbb{P}_{\Vdash}(\mathfrak{R}) \qquad \Gamma \Vdash tl \cong tl' : X \mid \mathbb{P}_{\Vdash}(\mathfrak{R})}{\Gamma \Vdash_{\mathrm{nf}} hd ::_P tl \cong hd' ::_{P'} tl' : X \mid \mathfrak{R}}$$

$$\textsc{NilRed}\ \frac{\Gamma \Vdash \mathbb{P}(\mathfrak{R}) \cong P \mid \mathbb{P}_{\Vdash}(\mathfrak{R}) \\ \Gamma \Vdash \mathbb{P}(\mathfrak{R}) \cong P' \mid \mathbb{P}_{\Vdash}(\mathfrak{R})}{\Gamma \Vdash_{\mathrm{nf}} \varepsilon_P \cong \varepsilon_{P'} : X \mid \mathfrak{R}} \qquad \textsc{NeRed}\ \frac{\Gamma \vdash_{\mathrm{map}} n \approx n' : \mathbf{List}\,\mathbb{P}(\mathfrak{R})}{\Gamma \Vdash_{\mathrm{nf}} n \cong n' : X \mid \mathfrak{R}}$$

$$\textsc{MapNeConvRedL}\ \frac{\Gamma \vdash_{\mathrm{map}} n \approx n' : \mathbf{List}\,\mathbb{P}(\mathfrak{R}) \\ \Gamma, x{:}\, \mathbb{P}(\mathfrak{R}) \Vdash f\,x \cong x : \mathbb{P}(\mathfrak{R}) \mid \mathbb{P}_{\Vdash}(\mathfrak{R})}{\Gamma \Vdash_{\mathrm{nf}} \mathrm{map}_{\mathbf{List}}\ f\,n \cong n' : X \mid \mathfrak{R}} \qquad \textsc{NeMapConvRedR}\ \ldots$$

$$\textsc{MapMapConvRed}\ \frac{\Gamma \vdash_{\mathrm{map}} n \approx n' : \mathbf{List}\,A \qquad \Gamma, x{:}\, A \Vdash f\,x \cong f'\,x : \mathbb{P}(\mathfrak{R}) \mid \mathbb{P}_{\Vdash}(\mathfrak{R})}{\Gamma \Vdash_{\mathrm{nf}} \mathrm{map}_{\mathbf{List}}\ f\,n \cong \mathrm{map}_{\mathbf{List}}\ f'\,n' : X \mid \mathfrak{R}}$$

Fig. 8. Reducible convertibility of lists (where $\mathfrak{R}$ is a proof of $\Gamma \Vdash_{\mathbf{List}} X$)

Validity of the Functor Laws All the expected properties extend to this new logical relation: reflexivity, symmetry, transitivity, irrelevance with respect to reducible conversion, stability by weakening and anti-reduction.[11] These properties are essential in order to show that the logical relation validates the functor laws on any reducible term. The proof proceeds through an usual argument for logical relations: on canonical forms, the functor laws hold as observed already in Section 3.3; on compacted neutrals and neutral forms, we need to show that any compositions of $\mathrm{map}_{\mathbf{List}}$ reduce to a single map of a function with a reducible body, which amounts to show that composing reducible functions produces reducible outputs on reducible inputs. This last step in the proof reflect our assumption that the categorical structure equipping domains of type formers, here dom(**List**), should be definitionally associative and unital.

[11] Available in the directory LogicalRelation.

4.2 Deciding Conversion and Typechecking for $\mathrm{MLTT}_{\mathrm{map}}$

Instantiating the generic typing interface of the logical relation with declarative typing provides metatheoretic consequences of the existence of normal forms, among which normalization, injectivity of type constructors and subject reduction. Using those, we can show that algorithmic typing is sound directly by induction, and also that it fits the generic typing interface of the logical relation, which lets us derive that it is complete with respect to declarative typing.

This part of the proof is close to Abel et al. [2] and Adjedj et al. [3]. The main change is that we adapt algorithmic conversion to reflect the addition of compacted neutrals in our definition of normal forms, by introducing a third mutually defined relation to compare these compacted neutrals. The main idea is summed up in rules LNConv and LNMap below: when comparing compacted neutrals, we use the new relation $\approx_{\mathrm{map}}$, which simulates the behaviour of the logical relation from Figure 8 on compacted neutrals.

$$\textsc{LNConv}\ \frac{\Gamma \vdash_{\mathrm{map}} c \approx_{\mathrm{map}} c' \lhd \mathbf{List}\, A}{\Gamma \vdash_{\mathrm{map}} c \cong_{\mathrm{h}} c' \lhd \mathbf{List}\, A} \qquad \textsc{LNMap}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{map}} n \approx_{\mathrm{h}} n' \rhd \mathbf{List}\, A \\ \Gamma, x\colon A \vdash_{\mathrm{map}} f\, x \cong x \lhd B\end{array}}{\Gamma \vdash_{\mathrm{map}} \mathrm{map}_{\mathbf{List}}\, f\, n \approx_{\mathrm{map}} n' \lhd \mathbf{List}\, B}$$

Using this second, algorithmic, instance as a specification, we can show the soundness and completeness of a conversion-checking function extending that of Adjedj et al. [3] with lists and neutral compaction. Thus, via the equivalence of declarative and algorithmic conversion, we obtain decidability of the rich equational theory of (declarative) $\mathrm{MLTT}_{\mathrm{map}}$.

5 Subtyping, Coercive and Subsumptive

The main application we develop for our definitional functor laws is structural subtyping. More precisely, we describe two extensions of MLTT. The first, $\mathrm{MLTT}_{\mathrm{sub}}$, has subsumptive subtyping: whenever $\vdash_{\mathrm{sub}} t : A \preccurlyeq A'$, then also $\vdash_{\mathrm{sub}} t : A'$, leaving subtyping implicit. The second, $\mathrm{MLTT}_{\mathrm{coe}}$, features coercive subtyping, witnessed by an operator $\mathrm{coe}_{A,A'}\, t$ explicitly marking where subtyping is used and well-typed whenever $\vdash_{\mathrm{coe}} t : A \preccurlyeq A'$. The computational behaviour of coe on each type former is informed by the corresponding map in $\mathrm{MLTT}_{\mathrm{map}}$. Structural coercions can hence be studied modularly in $\mathrm{MLTT}_{\mathrm{map}}$ and tied together in $\mathrm{MLTT}_{\mathrm{coe}}$.

In Section 5.1, we give algorithmic presentations of $\mathrm{MLTT}_{\mathrm{coe}}$ and $\mathrm{MLTT}_{\mathrm{sub}}$. In the context of a proof assistant or dependently typed programming language, $\mathrm{MLTT}_{\mathrm{sub}}$ would be the flexible, user-facing system, and $\mathrm{MLTT}_{\mathrm{coe}}$ its well-behaved specification. We do not develop the equivalence between this algorithmic presentation of $\mathrm{MLTT}_{\mathrm{coe}}$ and its declarative variant, as its proof is similar to the one for $\mathrm{MLTT}_{\mathrm{map}}$.

Section 5.2 relates $\mathrm{MLTT}_{\mathrm{coe}}$ and $\mathrm{MLTT}_{\mathrm{sub}}$: there is a simple erasure $|\cdot|$ from the former to the latter which removes coercions, and we show it is type-preserving; conversely, we show that any well-typed $\mathrm{MLTT}_{\mathrm{sub}}$ term can be elaborated to a well-typed $\mathrm{MLTT}_{\mathrm{coe}}$ term. The extra definitional functor laws are essential at this stage, to ensure that all equalities valid in $\mathrm{MLTT}_{\mathrm{sub}}$ still hold in $\mathrm{MLTT}_{\mathrm{coe}}$. Since we are in a dependently typed system, if equations valid in $\mathrm{MLTT}_{\mathrm{sub}}$ failed to hold in $\mathrm{MLTT}_{\mathrm{coe}}$, elaboration could not be type-preserving. Finally, Section 5.3 discusses the implications of this equivalence for coherence.

5.1 The Type Systems $\mathrm{MLTT}_{\mathrm{sub}}$ and $\mathrm{MLTT}_{\mathrm{coe}}$

We focus on the structural aspect of subtyping, and a base case would be needed to have a non-trivial subtyping relation, i.e. to relate more types than conversion. We do not present a base case for subtyping due to space constraints, but refinement types with subtyping induces by the implication orders on formulas or record types with width and depth subtyping are typical instances for such base case. The latter is presented in [32].

$\boxed{\Gamma \vdash_{\mathrm{sub}} T \preccurlyeq_{\mathrm{h}} T' \lhd}$ Reduced type T is a subtype of reduced type T'

$$\textsc{UniSub}\ \frac{}{\Gamma \vdash_{\mathrm{sub}} \mathrm{Type}_i \preccurlyeq_{\mathrm{h}} \mathrm{Type}_i \lhd} \qquad \textsc{ProdSub}\ \frac{\Gamma \vdash_{\mathrm{sub}} A' \preccurlyeq A \lhd \quad \Gamma, x{:}\, A' \vdash_{\mathrm{sub}} B \preccurlyeq B' \lhd}{\Gamma \vdash_{\mathrm{sub}} \Pi\, x{:}\, A.B \preccurlyeq_{\mathrm{h}} \Pi\, x{:}\, A'.B' \lhd}$$

$$\textsc{ListSub}\ \frac{\Gamma \vdash_{\mathrm{sub}} A \preccurlyeq A' \lhd}{\Gamma \vdash_{\mathrm{sub}} \mathbf{List}\, A \preccurlyeq_{\mathrm{h}} \mathbf{List}\, A' \lhd} \qquad \textsc{NeuSub}\ \frac{\Gamma \vdash_{\mathrm{sub}} n \approx_{\mathrm{h}} n' \rhd T}{\Gamma \vdash_{\mathrm{sub}} n \preccurlyeq_{\mathrm{h}} n' \lhd}$$

Fig. 9. Algorithmic subtyping between reduced types (extends Figure 3)

Algorithmic $\mathrm{MLTT}_{\mathrm{sub}}$ This system replaces Check with the following rule, which uses subtyping $\preccurlyeq$ instead of conversion:

$$\textsc{CheckSub}\ \frac{\Gamma \vdash_{\mathrm{sub}} t \rhd T' \qquad \Gamma \vdash_{\mathrm{sub}} T' \preccurlyeq T \lhd}{\Gamma \vdash_{\mathrm{sub}} t \lhd T}$$

Subtyping, defined in Figure 9, orients type-level conversion from Figure 3, taking into account co- and contravariance. It relies on neutral comparison and term-level conversion, both of which are *not* altered with respect to Figure 3: subtyping is a type-level concept only.

Algorithmic $\mathrm{MLTT}_{\mathrm{coe}}$ In contrast with $\mathrm{MLTT}_{\mathrm{sub}}$, rule Check in $\mathrm{MLTT}_{\mathrm{coe}}$ is *not* altered. Instead, subtyping is only allowed when *explicitly* marked by coe, as follows:

$$\textsc{Coe}\ \frac{\Gamma \vdash_{\mathrm{coe}} A \lhd \qquad \Gamma \vdash_{\mathrm{coe}} A' \lhd \qquad \Gamma \vdash_{\mathrm{coe}} t \lhd A \qquad \Gamma \vdash_{\mathrm{coe}} A \preccurlyeq A' \lhd}{\Gamma \vdash_{\mathrm{coe}} \mathrm{coe}_{A,A'}\, t \rhd A'}$$

$\boxed{t \rightsquigarrow^1 t'}$

$$\frac{\mathrm{nf}\, f}{(\mathrm{coe}_{\Pi\, x{:}A'.B',\Pi\, x{:}A.B}\, f)\ a \rightsquigarrow^1 \mathrm{coe}_{B'\left[\mathrm{coe}_{A,A'}\, a\right],B[a]}(f\ (\mathrm{coe}_{A,A'}\, a))} \qquad \mathrm{coe}_{\mathrm{Type}_i,\mathrm{Type}_i}\, t \rightsquigarrow^1 t$$

$$\mathrm{coe}_{\mathbf{List}\, A,\mathbf{List}\, A'}\, \varepsilon \rightsquigarrow^1 \varepsilon \qquad \mathrm{coe}_{\mathbf{List}\, A,\mathbf{List}\, A'}(h :: t) \rightsquigarrow^1 \mathrm{coe}_{A,A'}\, h :: \mathrm{coe}_{\mathbf{List}\, A,\mathbf{List}\, A'}\, t$$

$$\textsc{CoeL}\ \frac{A \rightsquigarrow^1 A'}{\mathrm{coe}_{A,B}\, t \rightsquigarrow^1 \mathrm{coe}_{A',B}\, t} \qquad \textsc{CoeR}\ \frac{\mathrm{nf}^{\oplus}\ \text{or ne}\ A \qquad B \rightsquigarrow^1 B'}{\mathrm{coe}_{A,B}\, t \rightsquigarrow^1 \mathrm{coe}_{A,B'}\, t}$$

$$\textsc{CoeTm}\ \frac{\mathrm{nf}^{\oplus}\ \text{or ne}\ A, B \qquad t \rightsquigarrow^1 t'}{\mathrm{coe}_{A,B}\, t \rightsquigarrow^1 \mathrm{coe}_{A,B}\, t'} \qquad \textsc{CoeCoe}\ \frac{\mathrm{nf}^{\oplus}\ \text{or ne}\ U, U', T, T' \qquad \mathrm{ne}\, n}{\mathrm{coe}_{U,U'}\, \mathrm{coe}_{T,T'}\, n \rightsquigarrow^1 \mathrm{coe}_{T,U'}\, n}$$

$\boxed{\mathrm{nf} \quad f}$	$\overset{\text{def}}{=} n \mid P \mid N \mid \lambda\, x{:}\, t.t \mid \varepsilon_t \mid t ::_t t \mid \mathrm{coe}_{N,N}\, f \mid \ldots$	weak-head normal forms
$\boxed{\mathrm{nf}^{\ominus} \quad N}$	$\overset{\text{def}}{=} \Pi\, x{:}\, t.t \mid \Sigma\, x{:}\, t.t$	negative whnf types
$\boxed{\mathrm{nf}^{\oplus} \quad P}$	$\overset{\text{def}}{=} \mathrm{Type}_i \mid \mathbf{List}\, t \mid \ldots$	positive whnf types
$\boxed{\mathrm{ne} \quad n}$	$\overset{\text{def}}{=} x \mid n\ t \mid n.l \mid \mathrm{ind}_P(c; t; t) \mid \ldots$	weak-head neutrals
$\boxed{\mathrm{cne} \quad c}$	$\overset{\text{def}}{=} n \mid \mathrm{coe}_{P,P}\, n \mid \mathrm{coe}_{n,n}\, n$	compacted neutrals

Fig. 10. Weak-head reduction rules for coercion (extends Figure 2)

Reduction must of course be extended to give an operational behaviour to coe, and is given in Figure 10, together with normal forms. Operationally, $\mathrm{coe}_{A,A'}\, t$ reduces the types A and A' to head normal forms, then behaves like the relevant map, propagating coe recursively. Since $\mathrm{coe}_{A,A'}\, t$ is well-typed only when A is a subtype of A', the type formers of their head normal forms have to agree, ensuring that we can always rely on this behaviour to enact structural subtyping. As for map, rule CoeCoe lets us compact a succession of stuck coe. This only applies to positive types (characterized by $\mathrm{nf}^{\oplus}$): we do not compact coercions between negative/extensional types, but wait for the term to be observed to trigger further reduction.

Neutral conversion is described at the top of Figure 11 and features an additional comparison between compacted neutrals similar to $\mathrm{MLTT}_{\mathrm{map}}$ (LNConv). Rule NCoe is a congruence for coercions, where the source and target types necessarily agree by typing invariants, and are thus not compared. Rules NCoeL and NCoeR handle identity coercions. Accordingly, $\approx_{\mathrm{coe}}$ is carefully used whenever normal forms can be compacted neutrals, *e.g.* at neutral and positive types, as shown at the bottom of Figure 11. Apart from this change, conversion at the term and type level and subtyping are similar to those of $\mathrm{MLTT}_{\mathrm{sub}}$.

5.2 Elaboration and Erasure

We can now turn to the correspondence between $\mathrm{MLTT}_{\mathrm{sub}}$ and $\mathrm{MLTT}_{\mathrm{coe}}$. The translation in the forward direction, *erasure* $|\cdot|$, removes coercions $\left|\mathrm{coe}_{A,A'}\, t\right| = t$

$\boxed{\Gamma \vdash_{\text{coe}} t \approx_{\text{coe}} t' \lhd T}$ Compacted neutrals t and t' are comparable at type T

$$\text{NCoe}\ \frac{\Gamma \vdash_{\text{coe}} n \approx n' \rhd S''}{\Gamma \vdash_{\text{coe}} \text{coe}_{S,T}\, n \approx_{\text{coe}} \text{coe}_{S',T'}\, n' \lhd T''} \qquad \text{NCoeL}\ \frac{\Gamma \vdash_{\text{coe}} n \approx n' \rhd S''}{\Gamma \vdash_{\text{coe}} \text{coe}_{S,T}\, n \approx_{\text{coe}} n' \lhd T''}$$

$$\text{NCoeR}\ \frac{\Gamma \vdash_{\text{coe}} n \approx n' \rhd S''}{\Gamma \vdash_{\text{coe}} n \approx_{\text{coe}} \text{coe}_{S',T'}\, n' \lhd T''} \qquad \text{NNoCoe}\ \frac{\Gamma \vdash_{\text{coe}} n \approx n' \rhd S''}{\Gamma \vdash_{\text{coe}} n \approx_{\text{coe}} n' \lhd T''}$$

$\boxed{\Gamma \vdash_{\text{coe}} t \cong_{\text{h}} t' \lhd T}$

$$\text{NeuList}\ \frac{\Gamma \vdash_{\text{coe}} n \approx_{\text{coe}} n' \lhd \textbf{List}\, A}{\Gamma \vdash_{\text{coe}} n \cong_{\text{h}} n' \lhd \textbf{List}\, A} \qquad \text{NeuNeu}\ \frac{\Gamma \vdash_{\text{coe}} n \approx_{\text{coe}} n' \lhd M \qquad \text{ne}\, M}{\Gamma \vdash_{\text{coe}} n \cong_{\text{h}} n' \lhd M}$$

Fig. 11. Algorithmic comparison of neutrals in MLTT_{coe} (extends Figure 3)

and is otherwise a congruence. It is lifted pointwise to contexts. We first show that erasure is sound, meaning that it preserves typing and conversion, and then that it is also invertible, *i.e.* that any well-typed MLTT_{sub} term t' elaborates to a well-typed MLTT_{coe} term t whose erasure is $t' = |t|$.

Soundness of Erasure Erasure translates from a constrained system to a more liberal one. Establishing its soundness, *e.g.*that conversion and typing are preserved, is relatively easy, as long as the reduction rules of Figure 10 are designed so that erasure preserves them. Indeed, the key point is that reduction rules for coe do *not* change the structure of the erased term, and so erase to exactly zero steps of reduction. In contrast, the rule below is inadequate, as it would η-expand terms at function types more in MLTT_{coe} than in MLTT_{sub}:

$$\text{coe}_{\Pi\, x:A'.B', \Pi\, x:A.B}\, f \rightsquigarrow^1 \lambda\, x\colon A.\, \text{coe}_{B'[\text{coe}_{A,A'}\, x], B}(f\ \ \text{coe}_{A,A'}\, x).$$

The two terms remain nonetheless convertible. By induction on MLTT_{coe}'s typing derivation, one can then show that erasure preserves conversion and subtyping, and finally typing.

Theorem 1 (Erasure Preserves Typing). *If* $\Gamma \vdash_{\text{coe}} t \lhd T$ *holds and its inputs are well-formed, then* $|\Gamma| \vdash_{\text{sub}} |t| \lhd |T|$.

Elaboration Elaborating back from MLTT_{sub} to MLTT_{coe} is more challenging: as we add annotations, we must ensure that these do not hinder conversion. We follow the proof strategy of a similar proof of elaboration soundness in Lennon-Bertrand et al. [33]. The core of the argument are so-called "catch-up lemmas", which ensure that annotations never block redexes. As an example, here is the one for function types.

Lemma 2 (Catch up, Function Type). *If* $\Gamma \vdash_{\text{coe}} f\ a : B$ *and* $|f| = \lambda\, x\colon A'.\, t'$, *then there exists* t *such that* $|t| = t'$ *and* $f\ a \rightsquigarrow^\star t[a]$.

From these catch-up lemmas it follows that erasure is a backward simulation, therefore that it preserves subtyping, and finally that it is type-preserving. Proofs are all by induction, and given in [32].

Lemma 3 (Erasure is a Backward Simulation). *Assume that* $\Gamma \vdash_{\mathrm{coe}} t : T$*. If* $|t| \rightsquigarrow^{\star} u'$*, with* u' *a weak-head normal form, then* $t \rightsquigarrow^{\star} u$*, with* u *a weak-head normal form such that* $|u| = u'$*.*

Lemma 4 (Elaboration Preserves Subtyping). *The following implications hold whenever the inputs of the conclusions are well-formed:*

1. *if* $|\Gamma| \vdash_{\mathrm{sub}} |T| \preccurlyeq |U| \lhd$*, then* $\Gamma \vdash_{\mathrm{coe}} T \preccurlyeq U \lhd$*;*
2. *if* $|\Gamma| \vdash_{\mathrm{sub}} |t| \cong |u| \lhd |T|$*, then* $\Gamma \vdash_{\mathrm{coe}} t \cong u \lhd T$*;*
3. *if* $|\Gamma| \vdash_{\mathrm{sub}} |t| \approx |u| \rhd T$*, then* $\Gamma \vdash_{\mathrm{coe}} t \approx u \rhd T$*;*
4. *and similarly for the other judgements.*

Finally, the main theorem states that we can elaborate terms using implicit subtyping to explicit coercions, in a type-preserving way.

Corollary 1 (Elaboration). *If* $\vdash_{\mathrm{coe}} \Gamma$*,* $\Gamma \vdash_{\mathrm{coe}} T \lhd$ *and* $|\Gamma| \vdash_{\mathrm{sub}} t' \lhd |T|$*, then there exists* t *such that* $\Gamma \vdash_{\mathrm{coe}} t \lhd T$*, and* $|t| = t'$*.*

Importantly, to establish this equivalence we do *not* need to develop any meta-theory for $\mathrm{MLTT}_{\mathrm{sub}}$: having the meta-theory of $\mathrm{MLTT}_{\mathrm{coe}}$ is enough!

Nonetheless, now that the equivalence between the two systems has been established, we can use it to transport meta-theoretic properties, such as normalization, from $\mathrm{MLTT}_{\mathrm{coe}}$ to $\mathrm{MLTT}_{\mathrm{sub}}$.

5.3 Coherence

An important property of elaboration is *coherence*, stating that the elaboration of a well-typed term does not depend on its typing derivation. In our algorithmic setting, a term has at most one typing derivation and so at most one elaboration. However, multiple well-typed terms in $\mathrm{MLTT}_{\mathrm{coe}}$ can still erase to the same $\mathrm{MLTT}_{\mathrm{sub}}$ term. While only one of them is the result of elaboration as defined in Corollary 1, all these distinct terms should still behave similarly. The following is a direct consequence of Lemma 4, and shows that the equations imposed on coe are enough to give us a very strong form of coherence: it holds up to definitional equality, rather than in a weaker, semantic way. Another way to look at this is that the scenario of Example 2 cannot happen, thanks to our new equations: if two terms erase to the same coercion-free one in $\mathrm{MLTT}_{\mathrm{sub}}$, then they *must* be convertible in $\mathrm{MLTT}_{\mathrm{coe}}$. Hidden coercions cannot be responsible for failures of conversion.

Theorem 2 (Coherence). *If* t*,* u *are such that* $\Gamma \vdash_{\mathrm{coe}} t \lhd T$ *and* $\Gamma \vdash_{\mathrm{coe}} u \lhd T$*, with* $\vdash_{\mathrm{coe}} \Gamma$ *and* $\Gamma \vdash_{\mathrm{coe}} T \lhd$*, and moreover* $|t| = |u|$ *(i.e. t and u correspond to the same* $\mathit{MLTT}_{\mathrm{sub}}$ *term), then* $\Gamma \vdash_{\mathrm{coe}} t \cong u \lhd T$*.*

Proof. By reflexivity, (obtained through the equivalence with the declarative system), $\Gamma \vdash_{\text{coe}} t \cong t \lhd T$. Using Theorem 1 (soundness of erasure), we get $|\Gamma| \vdash_{\text{sub}} |t| \cong |t| \lhd |T|$, and so also $|\Gamma| \vdash_{\text{sub}} |t| \cong |u| \lhd |T|$. But then by Lemma 4 (elaboration preserving conversion), we can come back, and obtain $\Gamma \vdash_{\text{coe}} t \cong u \lhd T$.

As particular cases of this coherence theorem, we can now exhibit the necessity of the functor laws, sharpening the informal argument in the introduction. For the identity law, any well-typed MLTT_{coe} term $\text{coe}_{A,A}\, t$ erases to $\left|\text{coe}_{A,A}\, t\right| = |t|$ in MLTT_{sub}, and by coherence we obtain that the conversion $\Gamma \vdash_{\text{coe}} \text{coe}_{A,A}\, t \cong t \lhd A$ is required in MLTT_{coe}. For the composition law, we have for adequately well-typed terms that $\left|\text{coe}_{B,C}\, \text{coe}_{A,B}\, t\right| = |t| = \left|\text{coe}_{A,C}\, t\right|$, hence by coherence the conversion $\Gamma \vdash_{\text{coe}} \text{coe}_{B,C}\, \text{coe}_{A,B}\, t \cong \text{coe}_{A,C}\, t \lhd$ must hold in MLTT_{coe} as well.

6 Related and Future Work

Adding Definitional Equations to Dependent Type Theory Strub [50] endows a dependent type theory with additional equations from first order decidable theories, with further extensions to a universe hierarchy and large eliminations in Jouannaud et al. [28] and Barras et al. [11]. Equational theories can sometimes be presented by a confluent set of rewrite rules, a case advocated by Cockx et al. [17]. They show through counter-examples that ensuring type preservation in dependent type theory is a subtle matter and do not ensure normalization of the resulting theory. On the theoretical side, categorical tools are being developed to prove general conservativity and strictification results for type theories [12, 13] extending the seminal work of Hofmann [24] on conservativity of extensional type theory with respect to intensional type theory [57].

Formalized Metatheory with Logical Relations. Allais et al. [6] propose to add a variety of fusion laws for lists, including our functor laws, to a simply typed λ-calculus, only sketching an extension to dependent types. The three classes of normal forms (see Figures 7 and 10) is inspired from their work. While we depart from their normalization by evaluation approach to obtain fine-grained results on convergence of iterated weak-head reduction, we expect that the original strategy should extend to dependent types. Formalizing logical relations for MLTT is a difficult exercise, pioneered by Abel et al. [2] in AGDA using inductive-recursive definitions, and Wieczorek et al. [55] in COQ using impredicativity. We build upon and extend a COQ reimplementation of the former [3].

Cast and Coercion Operators Pujet et al. [45, 46] extend Abel et al. [2] to establish the metatheory of observational type theory [8]. Their work features a `cast` operator behaving similarly to coe, but guarded by an internal proof of equality instead of an external subtyping derivation. Their `cast` does not satisfy definitional transitivity, and we give evidence in [32] that such an extension would break metatheoretical properties. Another cast primitive with a similar operational behaviour appears in cast calculi for gradual typing [47], and indeed our proof that elaboration is type preserving in Section 5.2 is inspired by a similar one for GCIC, which combines gradual and dependent types [33]. In this case, casting is allowed between any two types, but the absence of guard is compensated by the possibility of runtime errors, making the type theory inconsistent.

Functorial Maps for Inductive Type Schemes Luo et al. [36] describe the construction of map for a class of strictly positive operators on paper, but do not implement it. Deriving map-like construction is a typical example of metaprogramming frameworks for proof assistants, *e.g.* COQ-ELPI [19, 52] in COQ, and the generics AGDA library [21] derives a fold operation, from which map can be easily obtained. In a simply typed setting, Barral et al. [10] employ rewriting techniques, in particular rewriting postponement, to show that an oriented variant of the functor laws are confluent and normalizing. These techniques rely on normalization, and could not be easily adapted to the dependent setting, however the idea of postponing the reduction step for identity appears in our logical relation as well. In a short abstract, McBride et al. [42] investigate a notion of functorial adapters that generalizes and unifies both the CHECK rule from bidirectional typing and the COE rule from $\mathrm{MLTT}_{\mathrm{coe}}$.

Subtyping, Dependent Types and Algorithmic Derivations Coherence of coercions in presence of structural subtyping is a challenging problem. To address the issue, Luo et al. [37] introduce a notion of weak transitivity, weakening the coherence of the transitivity up to propositional equality. This solution does not interact well with dependency, forcing them to restrict structural subtyping to a class of non-dependent inductives, *e.g.* excluding (positive) Σ. Luo et al. [36] show that the transitivity of coercions is admissible in presence of definitional compositions – called χ-rules there – for inductive schemata. They rely on a conjecture that strong normalization and subject reduction hold in presence of these χ-rules, explicitly mentioning that the metatheory with those additional equality rules is "largely unknown". We provide such results, and have formalized them for **List**. We use a completely different proof technique, that scales to a theory with universes and large elimination. Both aforementioned papers employ a strict order for subtyping and do not consider the functor law for the identity, nor tackle decidability of type-checking.

Aspinall et al. [9] investigate the relationship between subtyping and dependent types using algorithmic derivations to control the subtyping derivations for a variant of λP, a type theory logically much weaker than MLTT. Lungu et al. [35] study an elaboration of a subsumptive presentation into coercive one in presence of a coherent signature of subtyping relations between base types. Assuming normalization, they show that subtyping extends to Π types, setting aside other parametrized types. While they work over an abstract signature of coercions, the functor laws we study are needed to instantiate this signature with meaningful datatypes while respecting their assumptions. We explain the relation of these algorithmic system with bidirectional systems, notably the one of Abel et al. [2], contributing to a sharper picture.

Integration with Other Forms of Subtyping As we mentioned in Section 5, our design of base subtyping was guided by simplicity. Our work on structural subtyping should integrate mostly seamlessly with other, more ambitious forms of subtyping. Coercions between dependent records form the foundation of hierarchical organizations of mathematical structures [4, 18, 56], and should be a simple extension of our framework. This could lead to vast simplification of the complex apparatus currently needed to deal with these hierarchies.

Refinement subtyping is heavily used in F* but also in Coq's Program [48] to specify the behaviour of programs. Relativizing any result of decidability of type-checking to that of the chosen fragment of refinements, an implementation of refinement subtyping using definitionally irrelevant propositions [23] to preserve coherence[12] should be within reach.

Our techniques for structural subtyping should also apply well in the context of algebraic approaches to cumulativity between universes [29, 49]. Cumulativity goes beyond mere subtyping, as it also involves definitional isomorphisms between two copies of the same type at different universe levels. Our definitional functor laws already allow these to interact well with map operations, but it would be interesting to investigate which extra definitional equations are needed – and can be realized – to make structural cumulativity work seamlessly, hopefully obtaining a translation from Russel-style to Tarski-style universes similar to our elaboration from $\mathrm{MLTT}_{\mathrm{sub}}$ to $\mathrm{MLTT}_{\mathrm{coe}}$.

Data Availability Statement An archive containing the formalization presented in Section 4 is available at [31]. The Coq code is supplemented with a report and a Docker image.

Acknowledgments The authors thank the anonymous reviewers for their feedback. The many useful discussions with Assia Mahboubi, Enzo Crance, Nicolas Tabareau and Loïc Pujet helped a lot to develop the material that led to this paper.

References

[1] Michael Gordon Abbott et al. "Containers: Constructing strictly positive types". In: *Theor. Comput. Sci.* 342.1 (2005), pp. 3–27. DOI: `10.1016/j.tcs.2005.06.002`. URL: `https://doi.org/10.1016/j.tcs.2005.06.002`.

[12] With proof-relevant propositions, different proofs of $p \Rightarrow q$ induce different coercions between $\{x \mid p\}$ and $\{x \mid q\}$, breaking coherence.

[2] Andreas Abel et al. “Decidability of Conversion for Type Theory in Type Theory”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158111.

[3] Arthur Adjedj et al. “Martin-Löf à la Coq”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024.* 2024.

[4] Reynald Affeldt et al. “Competing inheritance paths in dependent type theory: a case study in functional analysis”. In: *IJCAR 2020 - International Joint Conference on Automated Reasoning.* Paris, France, June 2020, pp. 1–19. URL: https://inria.hal.science/hal-02463336.

[5] Agda Development Team. *Agda 2.6.3 documentation.* 2023. URL: https://agda.readthedocs.io/en/v2.6.3/.

[6] Guillaume Allais et al. “New Equations for Neutral Terms: A Sound and Complete Decision Procedure, Formalized”. In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-Typed Programming.* DTP ’13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 13–24. ISBN: 9781450323840. DOI: 10.1145/2502409.2502411. URL: https://doi.org/10.1145/2502409.2502411.

[7] Thorsten Altenkirch et al. “Indexed containers”. In: *J. Funct. Program.* 25 (2015). DOI: 10.1017/S095679681500009X. URL: https://doi.org/10.1017/S095679681500009X.

[8] Thorsten Altenkirch et al. “Observational Equality, Now!” In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification.* PLPV ’07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 57–68. ISBN: 9781595936776. DOI: 10.1145/1292597.1292608.

[9] David Aspinall et al. “Subtyping dependent types”. In: *Theoretical Computer Science* 266.1 (2001), pp. 273–309. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(00)00175-4. URL: https://www.sciencedirect.com/science/article/pii/S0304397500001754.

[10] Freiric Barral et al. “Inductive Type Schemas as Functors”. In: *Computer Science - Theory and Applications, First International Symposium on Computer Science in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings.* Ed. by Dima Grigoriev et al. Vol. 3967. Lecture Notes in Computer Science. Springer, 2006, pp. 35–45. ISBN: 3-540-34166-8. DOI: 10.1007/11753728_7. URL: https://doi.org/10.1007/11753728%5C_7.

[11] Bruno Barras et al. “CoQMTU: A Higher-Order Type Theory with a Predicative Hierarchy of Universes Parametrized by a Decidable First-Order Theory”. In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada.* IEEE Computer Society, 2011, pp. 143–151. ISBN: 978-0-7695-4412-0. DOI: 10.1109/LICS.2011.37. URL: https://doi.org/10.1109/LICS.2011.37.

[12] Rafaël Bocquet. "Strictification of Weakly Stable Type-Theoretic Structures Using Generic Contexts". In: *27th International Conference on Types for Proofs and Programs, TYPES 2021, June 14-18, 2021, Leiden, The Netherlands (Virtual Conference).* Ed. by Henning Basold et al. Vol. 239. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 3:1–3:23. ISBN: 978-3-95977-254-9. DOI: `10.4230/LIPIcs.TYPES.2021.3`. URL: `https://doi.org/10.4230/LIPIcs.TYPES.2021.3`.

[13] Rafaël Bocquet. "Towards coherence theorems for equational extensions of type theories". In: *CoRR* abs/2304.10343 (2023). DOI: `10.48550/arXiv.2304.10343`. arXiv: `2304.10343`. URL: `https://doi.org/10.48550/arXiv.2304.10343`.

[14] Edwin C. Brady. "Idris 2: Quantitative Type Theory in Practice (Artifact)". In: *Dagstuhl Artifacts Ser.* 7.2 (2021), 10:1–10:7. DOI: `10.4230/DARTS.7.2.10`. URL: `https://doi.org/10.4230/DARTS.7.2.10`.

[15] Simon Castellan et al. "Undecidability of Equality in the Free Locally Cartesian Closed Category (Extended version)". In: *Log. Methods Comput. Sci.* 13.4 (2017). DOI: `10.23638/LMCS-13(4:22)2017`. URL: `https://doi.org/10.23638/LMCS-13(4:22)2017`.

[16] Jesper Cockx. *Disable all subtyping by default?* `https://github.com/agda/agda/issues/4474`. Accessed: 2023-07-10. 2020.

[17] Jesper Cockx et al. "The Taming of the Rew: A Type Theory with Computational Assumptions". In: *Proceedings of the ACM on Programming Languages.* POPL 2021 (2021). URL: `https://hal.archives-ouvertes.fr/hal-02901011`.

[18] Cyril Cohen et al. "Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi". In: *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction.* 167. Paris, France, June 2020, 34:1–34:21. DOI: `10.4230/LIPIcs.FSCD.2020.34`. URL: `https://inria.hal.science/hal-02478907`.

[19] Cvetan Dunchev et al. "ELPI: fast, Embeddable, λProlog Interpreter". In: *Proceedings of LPAR.* Suva, Fiji, Nov. 2015. URL: `https://inria.hal.science/hal-01176856`.

[20] Jana Dunfield et al. "Bidirectional Typing". In: *ACM Computing Surveys* 54.5 (May 2021). ISSN: 0360-0300. DOI: `10.1145/3450952`.

[21] Lucas Escot et al. "Practical Generic Programming over a Universe of Native Datatypes". In: *Proc. ACM Program. Lang.* 6.ICFP (2022). DOI: `10.1145/3547644`. URL: `https://doi.org/10.1145/3547644`.

[22] Lucas Escot et al. "Read the mode and stay positive". In: *29th International Conference on Types for Proofs and Programs.* 2023.

[23] Gaëtan Gilbert et al. "Definitional Proof-Irrelevance without K". In: *Proceedings of the ACM on Programming Languages.* POPL'19 3.POPL (Jan. 2019), pp. 1–28. DOI: `10.1145/329031610.1145/3290316`. URL: `https://hal.inria.fr/hal-01859964`.

[24] Martin Hofmann. *Extensional constructs in intensional type theory.* CPHC / BCS distinguished dissertations. Springer, 1997. ISBN: 978-3-540-76121-1.

[25] Cătălin Hrițcu. *Polarities: subtyping for datatypes.* https://github.com/FStarLang/FStar/issues/65. Accessed: 2023-07-04. 2014.

[26] Jasper Hugunin. "Why Not W?" In: *26th International Conference on Types for Proofs and Programs, TYPES 2020, March 2-5, 2020, University of Turin, Italy.* Ed. by Ugo de'Liguoro et al. Vol. 188. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 8:1–8:9. ISBN: 978-3-95977-182-5. DOI: 10.4230/LIPIcs.TYPES.2020.8. URL: https://doi.org/10.4230/LIPIcs.TYPES.2020.8.

[27] Bart P. F. Jacobs. *Categorical Logic and Type Theory.* Vol. 141. Studies in logic and the foundations of mathematics. North-Holland, 2001. ISBN: 978-0-444-50853-9. URL: http://www.elsevierdirect.com/product.jsp?isbn=9780444508539.

[28] Jean-Pierre Jouannaud et al. "Coq without Type Casts: A Complete Proof of Coq Modulo Theory". In: *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017.* Ed. by Thomas Eiter et al. Vol. 46. EPiC Series in Computing. EasyChair, 2017, pp. 474–489. DOI: 10.29007/bjpg. URL: https://doi.org/10.29007/bjpg.

[29] András Kovács. "Generalized Universe Hierarchies and First-Class Universe Levels". In: *30th EACSL Annual Conference on Computer Science Logic (CSL 2022).* Ed. by Florin Manea et al. Vol. 216. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 28:1–28:17. ISBN: 978-3-95977-218-1. DOI: 10.4230/LIPIcs.CSL.2022.28. URL: https://drops.dagstuhl.de/opus/volltexte/2022/15748.

[30] Meven Lennon-Bertrand. "Bidirectional Typing for the Calculus of Inductive Constructions". PhD thesis. Nantes Université, 2022.

[31] Meven Lennon-Bertrand et al. *Artifact for Definitional Functoriality for Dependent (Sub)Types.* 2024. URL: https://zenodo.org/records/10461809.

[32] Meven Lennon-Bertrand et al. *Definitional Functoriality for Dependent (Sub)Types.* 2023. URL: https://arxiv.org/abs/2310.14929.

[33] Meven Lennon-Bertrand et al. "Gradualizing the Calculus of Inductive Constructions". In: *ACM Transactions on Programming Languages and Systems* 44.2 (Apr. 2022). ISSN: 0164-0925. DOI: 10.1145/3495528.

[34] Miran Lipovača. *Learn You a Haskell for Great Good!* 2010. URL: http://learnyouahaskell.com/.

[35] Georgiana Elena Lungu et al. "On Subtyping in Type Theories with Canonical Objects". In: *22nd International Conference on Types for Proofs and Programs (TYPES 2016).* Ed. by Silvia Ghilezan et al. Vol. 97. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 13:1–13:31. ISBN: 978-3-95977-065-1. DOI: 10.4230/LIPIcs.TYPES.2016.13. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9849.

[36] Zhaohui Luo et al. "Structural subtyping for inductive types with functorial equality rules". In: *Mathematical Structures in Computer Science* 18.5 (2008), pp. 931–972. ISSN: 0960-1295. DOI: `10.1017/S0960129508006956`.

[37] Zhaohui Luo et al. "Transitivity in coercive subtyping". In: *Inf. Comput.* 197.1-2 (2005), pp. 122–144. DOI: `10.1016/j.ic.2004.10.008`. URL: `https://doi.org/10.1016/j.ic.2004.10.008`.

[38] Saunders MacLane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics, Vol. 5. New York: Springer-Verlag, 1971.

[39] Per Martin-Löf et al. *Intuitionistic Type Theory.* Studies in Proof Theory 1. Napoli: Bibliopolis, 1984.

[40] Conor McBride. "Grins from my Ripley Cupboard". 2009.

[41] Conor McBride. "Types Who Say Ni". 2022.

[42] Conor McBride et al. *Functorial Adapters.* 27th International Conference on Types for Proofs and Programs. June 2021.

[43] Leonardo de Moura et al. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings.* Ed. by André Platzer et al. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 625–635. ISBN: 978-3-030-79875-8. DOI: `10.1007/978-3-030-79876-5\_37`. URL: `https://doi.org/10.1007/978-3-030-79876-5%5C_37`.

[44] Benjamin C. Pierce et al. "Local Type Inference". In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. DOI: `10.1145/345099.345100`.

[45] Loïc Pujet et al. "Impredicative Observational Equality". In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: `10.1145/3571739`.

[46] Loïc Pujet et al. "Observational Equality: Now for Good". In: *Proc. ACM Program. Lang.* 6.POPL (2022). DOI: `10.1145/3498693`. URL: `https://doi.org/10.1145/3498693`.

[47] Jeremy G. Siek et al. "Refined Criteria for Gradual Typing". In: *1st Summit on Advances in Programming Languages (SNAPL 2015).* Ed. by Thomas Ball et al. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 274–293. DOI: `10.4230/LIPIcs.SNAPL.2015.274`.

[48] Matthieu Sozeau. "Subset Coercions in Coq". In: *Types for Proofs and Programs.* Ed. by Thorsten Altenkirch et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 237–252. ISBN: 978-3-540-74464-1.

[49] Jonathan Sterling. "Algebraic Type Theory and Universe Hierarchies". In: *CoRR* abs/1902.08848 (2019). arXiv: `1902.08848`.

[50] Pierre-Yves Strub. "Coq Modulo Theory". In: *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings.* Ed. by Anuj Dawar et al. Vol. 6247. Lecture Notes in Computer Science. Springer, 2010, pp. 529–543. ISBN: 978-3-642-15204-7. DOI: `10.1007/978-3-642-`

15205-4_40. URL: https://doi.org/10.1007/978-3-642-15205-4%5C_40.

[51] Nikhil Swamy et al. "Dependent types and multi-monadic effects in F". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík et al. ACM, 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837655. URL: https://doi.org/10.1145/2837614.2837655.

[52] Enrico Tassi. "Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)". working paper or preprint. Jan. 2018. URL: https://inria.hal.science/hal-01637063.

[53] The Coq Development Team. *The Coq Proof Assistant*. Version 8.16. Sept. 2022. DOI: 10.5281/zenodo.7313584. URL: https://doi.org/10.5281/zenodo.7313584.

[54] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[55] Paweł Wieczorek et al. "A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 266–279. ISBN: 9781450355865. DOI: 10.1145/3167091.

[56] Eric Wieser. *Multiple inheritance hazards in algebraic typeclass hierarchies*. 2023. arXiv: 2306.00617 [cs.LO].

[57] Théo Winterhalter et al. "Eliminating reflection from type theory". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi et al. ACM, 2019, pp. 91–103. ISBN: 978-1-4503-6222-1. DOI: 10.1145/3293880.3294095. URL: https://doi.org/10.1145/3293880.3294095.

Artifact Description
Definitional Functoriality for Dependent (Sub)Types

Théo Laurent[1], Meven Lennon-Bertrand[2], and Kenji Maillard[1](✉)

[1] Inria, Paris, France
{theo.laurent,kenji.maillard}@inria.fr
[2] University of Cambridge, Cambridge, United Kingdom
Meven.Lennon-Bertrand@cl.cam.ac.uk

1 Organization of the Artefact and Other Resources

This document describes the Coq formalisation accompanying the paper *Definitional Functoriality for Dependent (Sub)Types*, more specifically the content of section 4. To complement this document, we also provide the following:

– the `REQUIREMENTS.md` and `INSTALL.md` file with installation instructions;
– the `README.md` file with a quick overview of the development with hyperlinks to the files of interest;
– a `DOCKER.md` file, with installation and usage instructions for the provided docker image;
– a `Readme.v` file, which gives a more in-depth overview of the development as a Coq file, using directly the main Coq definitions and theorems, and is roughly similar to the present PDF description;
– a `doc/dependency_graph.png` file, showing the structure of the development.

We utilize the logical relation proof technique presented in Abel et al. [1] and build upon its Coq implementation due to Adjedj et al. [2]. This artefact contributes an extension of the formalisation with lists and definitional functor laws for lists. We refer to both articles for further details on the proof technique and the general setup of the formalisation.

2 Syntax

Terms (`AutoSubst/Ast`) The syntax of terms, along with the other files in the `AutoSubst` folder, are generated using the AutoSubst plugin. The definition of renaming and substitution are also automatically derived from the one of terms, and many boilerplate lemmas on them are too. Of particular interest are the constructors `tList`, `tNil`, `tCons`, `tElim` and `tMap`, respectively corresponding to the type constructor for lists, the empty list, list consing, the (dependent) eliminator for lists, and the definitionally functorial map operation.

`NormalForms` Weak-head normal forms `whnf`, neutrals `whne` and compacted neutrals `whne_list` are defined as inductive predicate on terms, *i.e.* as function of type `term -> Prop`, corresponding to Fig. 4 and 10 from the paper. In particular, any compacted neutral is a normal form, and compacted neutrals can either consist of a map of a neutral, or simply of a neutral.

Reduction (`UntypedReduction`*)* Reduction, written `[l ⇒* l']`, is the transitive closure of one-step reduction `[l ⇒ *']`, defined as an inductive relation. In particular, we have the rules of Fig. 9, that is:

```
mapNil : forall {A A' B f : term}, [tMap A B f (tNil A') ⇒ tNil B]
mapCons : forall {A A' B f a l : term},
  [tMap A B f (tCons A' a l) ⇒ tCons B (tApp f a) (tMap A B f l)]
mapComp : forall {A B B' C f g l : term},
  whne l -> [tMap B C f (tMap A B' g l) ⇒ tMap A C (comp A f g) l]
```

3 Typing and Conversion

`GenericTyping` Following Abel et al. [1] and Adjedj et al. [2], the definition of the logical relation is parametrized by a notion of *generic typing*, a common interface to be instantiated with both the declarative and algorithmic notions of typing. This interface features a family of judgments for context well-formation, typing, conversion but also a conversion of neutrals and a (typed) reduction relation. These judgements should satisfy properties, listed for each predicate with a record (`TypingProperties`, `ConvProperties`, etc.), and grouped together in the `GenericTypingProperties` record. We use type-classes to automatically find these properties when needed, and attach generic notations (defined in `Notations`) to these type-classes too.

For lists, generic typing closely resembles declarative typing, as defined in Fig. 2. Generic conversion must contain reduction, which includes typed variants of the rules above. Moreover, we have congruence rules for constructors, for instance we have the following, where `ta` stands for an arbitrary generic conversion:

```
forall (Γ : context) (A A' : term),
  [Γ |-[ ta ] A ≅ A' : U] -> [Γ |-[ ta ] tList A ≅ tList A' : U]
```

Conversion is not constrained to be a congruence for destructors, but it must contain neutral conversion, which *is* a congruence for `tMap` and `tListElim`, provided its main argument is too. Functor laws are also specified at the level of neutral conversion.

`DeclarativeTyping` The definition of the declarative judgments, as inductive predicates, corresponds to Fig. 2, 3, and 9 – the latter being restricted to the case of lists. The corresponding instance of generic typing is defined in `DeclarativeTypingInstance`. Neutral comparison is instantiated simply with conversion, *i.e.* the declarative instance does not distinguish between the two notions. Typed reduction is instantiated as the conjunction of declarative conversion and untyped reduction. All other judgments are directly instantiated with the corresponding declarative one.

`AlgorithmicTyping` The raw algorithmic typing judgments, akin to Fig. 5 and 6, are again defined as inductive predicates. As we explain at the end of Section

2.3 in relation to Fig. 7, we must impose extra pre-conditions for these judgments to be well-behaved. The corresponding judgments, called *bundled*, are defined in `BundledAlgorithmicTyping`. In `AlgorithmicConvProperties` and `AlgorithmicTypingProperties`, we establish the properties of the conversion and typing judgments, to derive two new instances of generic typing. The first instance uses (bundled) algorithmic conversion, but declarative typing. It depends on consequences of the logical relation instantiated with the fully declarative instance. The second uses only bundled algorithmic judgments, but depends on consequences of the logical relation instantiated with the first, mixed instance.

4 The Logical Relation

The logical relation is built from two layers, first the reducibility layer attaching witnesses of reducibility to weak-head normal form and second the validity layer that closes reducibility under substitution.

Definition of reducibility (`LogicalRelation`*)* The reducibility layer describes the types `A` that are reducible in a given context `Γ` and level `l`, noted `[Γ ||-<l> A ]`. Informally, a type is reducible when it weak-head reduces to a (weak-head) normal form, and the subterms of this normal form are themselves reducible. This weak-head normal form, when it exists, is unique by determinism of the weak-head reduction strategy. A witness of reducibility `RA : [Γ ||-<l> A ]` for the type `A` induce three subsequent predicates:

– reducible conversion of a type `B` to `A`, noted `[Γ ||-<l> A ≅ B| RA]`,
– reducibility of terms `t` of type `A`, noted `[Γ ||-<l> t : A | RA]`,
– reducible conversion of terms `t`,`u` of type `A`, noted `[Γ ||-<l> t ≅ u : A | RA]`.

These three predicates are packed in a single record `LRPack`. Reducible types are characterized inductively together with their associated `LRPack` using an indexed inductive `LR`. This encoding of a seemingly inductive-recursive definition using the inductively generated graph of the functions is known as small-induction recursion. The actual content of the reducibility relation is defined independently for each type formers as well as the neutrals types. We focus here on the reducibility of lists and refer to [2, 1] for the other type formers.

A type `A` is reducible as a list if it weak-head reduces to a type of shape `tList par` where the parameter type `par` is itself reducible in any weakening of the context `Γ`. This Kripke-style quantification on all future (weakened) contexts `Δ ≤ Γ` is necessary for specifying reducibility in larger contexts.

Reducible terms of list type are defined inductively in two steps: `ListProp` holds of canonical forms of type list (nil, cons and neutrals) with reducible arguments ; `ListRedTm` holds of terms that weak-head reduce to a reducible canonical form. The two inductive definitions must be mutual since the tail of a reducible `tCons` need not to be in weak-head normal form. A neutral term of list type is reducible if it is a well-typed neutral and moreover, if it is of shape `tMap A B f l` with `l` necessary neutral itself, then the body `f⟨wk1 Γ A⟩ (tRel 0)` of `f` must be reducible in an extended context `Γ,,A`. In the latter case, the type `B` of the codomain of `f` cannot be required to be reducible since that would lead to non-well-founded definition for the logical relation, but it is reducibly convertible to the reducible parameter type `par` at which reducibility of lists is defined.

Reducible conversion between terms of list type follow a similar pattern. In order to account for the identity functor law, the additional reducibility datum needed to relate two neutral terms also depends on the shape of the terms:

- if both terms are respectively of the shape `tMap A B f l` and `tMap A' B' f' l'`, then the bodies of `f` and `f'` must be reducibly convertible (congruence);
- if only one of the term is of shape `tMap A B f l`, then `f` must be reducibly convertible to the identity function, i.e. its body must be reducibly convertible to the first variable in context `tRel 0`.

Properties of Reducibility In order to reason on reducibility, we derive the induction principle corresponding to the inductive-recursive definition of the logical relation in `LogicalRelation/Induction`. This induction principle is then employed to derive a variety of properties of reducibility in the `LogicalRelation/` subdirectory: an inversion principle, irrelevance with respect to reducible conversion, reflexivity, symmetry and transitivity of reducible conversion, stability by weakening and by anti-reduction.

Validity and the Fundamental Lemma Validity closes reducibility by reducible substitution using another encoding of an inductive-recursive schema. The fundamental lemma then states that all components of a derivable declarative judgement are valid, in particular, terms well-typed for the declarative presentation are valid. The proof of the fundamental lemma proceed by an induction on declarative typing derivations, using that each declarative derivation step is admissible for the validity logical relation. These admissibility results are shown independently for each type former in the `Substitution/Introductions/` subdirectory. Most type and term formers related to lists are in `List`, while the eliminator for lists is in `ListElim`. The proofs follow the description of the logical relation: first, we show that each type, term or conversion equation is reducible using the definition and properties of reducibility, and then that it is valid. To show that the functor laws are valid, we use that composition of functions (e.g. morphisms for list) is definitionally associative and unital.

5 Type-checker (`Decidability` folder)

Open Recursion for Partial Functions To side-step issues with the complex termination argument of the conversion checker, we define it in an open recursion fashion, relying on a form of free monad. The functions for reduction, conversion and type checking are defined in `Decidability/Functions`. The main change compared to Adjedj et al. [2] is the addition of compaction to weak-head evaluation. Evaluation is implemented using a stack machine, on which elimination forms are pushed as they are encountered. When the machine hits a variable, for Adjedj et al. [2] it means the whole term – the variable against the stack of eliminations – is a neutral. However, this is not the case for us: we want to compute a compacted neutral. Thus, we add an extra compaction pass, implemented by the `compact` function, which merges successive map operations on the stack as we unpile them.

Correctness of the Functions Correctness of the implementations is shown in three steps. First, we show `Soundness`, *i.e.* that a positive answer of the checker implies the corresponding (algorithmic) judgment. Next, we show `Completeness`, *i.e.* that whenever an algorithmic judgment holds, then the corresponding checker answers positively. Finally, we show `Termination`, *i.e.* that the checkers always terminates when run on well-typed inputs. Again, the main innovation has to do with compaction. To reason about it, we need to make explicit the invariant that the stack is always "well-typed", in a suitable sense, see `typed_stack` in `Completeness`.

6 Main properties

The main properties we obtain from the logical relations and the certified checker are the following. First, every well-typed term and type are (weakly) normalising (proven in `Normalisation`):

```
Record WN (t : term) := {
  wn_val : term; wn_red : [ t ⇒* wn_val ]; wn_whnf : whnf wn_val; }.
Corollary normalisation {Γ A t} : [Γ |-[de] t : A] -> WN t.
Corollary type_normalisation {Γ A} : [Γ |-[de] A] -> WN A.
```

Conversion and typing are decidable (proven in `Decidability`):

```
Definition check_conv (Γ : context) (T t t' : term) (hΓ : [|- Γ])
  (hT : [Γ |- T]) (ht : [Γ |- t : T]) (ht' : [Γ |- t' : T]) :
  [Γ |- t ≅ t' : T] + ~[Γ |- t ≅ t' : T].
Definition check_full Γ (T t : term) : [Γ |- t : T] + ~[Γ |- t : T].
```

Finally, the type system seen as a logic is consistent, and canonicity holds at the type of natural numbers:

```
Lemma consistency {t} : [ε |- t : tEmpty] -> False.
Lemma nat_canonicity {t} : [ε |- t : tNat] ->
  ∑ n : nat, [ε |- t ≅ Nat.iter n tSucc tZero : tNat].
```

References

[1] Andreas Abel et al. "Decidability of Conversion for Type Theory in Type Theory". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: `10.1145/3158111`.

[2] Arthur Adjedj et al. "Martin-Löf à la Coq". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024*. 2024.

Author Index

S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 339–340, 2024.
https://doi.org/10.1007/978-3-031-57262-3