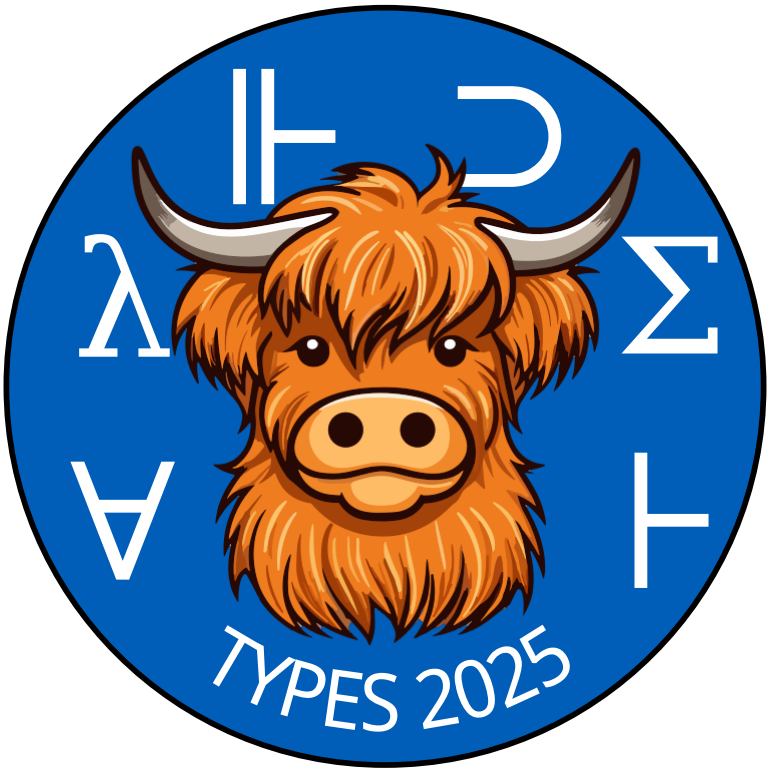
31st International Conference on
Types for Proofs and Programs

TYPES 2025

Abstracts



University of Strathclyde, Glasgow, Scotland, 9–13 June 2025

Jane Street
Well-Typed
The Haskell Consultants
Formal
VINDICATIONS
INPUT | OUTPUT
University of
Strathclyde
Glasgow

31st International Conference on Types for Proofs and Programs
TYPES 2025
University of Strathclyde, Glasgow, Scotland, 9–13 June 2025
Abstracts
https://msp.cis.strath.ac.uk/types2025/

Edited by Fredrik Nordvall Forsberg.

Department of Computer and Information Sciences
University of Strathclyde
Livingstone Tower
26 Richmond Street
Glasgow
G1 1XH

© 2025 the authors and the editor.

Preface

The TYPES meetings are a forum to present new and ongoing work in all aspects of type theory and its applications, especially in formalized and computer-assisted reasoning and computer programming. In the spirit of workshops, talks may be based on newly published papers, work submitted for publication, but also work in progress.

This year, 91 abstracts were submitted, and each submission was reviewed by at least three program committee members. Due to the large number of high-quality submissions and the limited space in the programme, it was decided by the programme committee to select some submissions for 20 minute talks, and some for 15 minute talks. In the end, 82 submissions were accepted (40 for a 20 minute slot, and 42 for a 15 minute slot).

The programme will also feature invited talks by Ingo Blechschmidt (University of Antwerp), Sonia Marin (University of Birmingham), Chris Martens (Northeastern University), Christian Sattler (Chalmers University of Technology and University of Gothenburg). One afternoon of the conference is co-located with the Women in EuroProofNet 2025 workshop, and will be dedicated to gender balance in the community.

The conference received financial support from the gold sponsor Jane Street, and the bronze sponsors Formal Vindications, Well-typed, and Input Output. It also received support in the form of caramel wafers from the local company Tunnock's. We gratefully acknowledge all support.

Similarly to previous years, a formal post-proceedings volume in Dagstuhl's *Leibniz International Proceedings in Informatics (LIPIcs)* series is planned for after the conference; papers submitted to that volume must not have been selected for presentation at the conference, but must represent unpublished work, and are subjected to (another) full peer-review process.

June 2025, Glasgow — Fredrik Nordvall Forsberg

Organisation

Programme Committee

Danel Ahman	University of Tartu, Estonia
Guillaume Allais	University of Strathclyde, Scotland
Malin Altenmüller	University of Edinburgh, Scotland
Sandra Alves	University of Porto, Portugal
Casper Bach	University of Southern Denmark, Denmark
Ana Bove	Chalmers University of Technology and University of Gothenburg, Sweden
Liang-Ting Chen	Academia Sinica, Taiwan
Vikraman Choudhury	University of Bologna, Italy
Gilda Ferreira	Universidade Aberta, Portugal
Daniel Gratzer	Aarhus University, Denmark
Tom de Jong	University of Nottingham, England
Dominik Kirst	Inria Paris, France
Neel Krishnaswami	University of Cambridge, England
András Kovács	University of Gothenburg and Chalmers University of Technology, Sweden
Peter LeFanu Lumsdaine	Stockholm University, Sweden
Kenji Maillard	Inria Rennes-Bretagne Atlantique, France
Max New	University of Michigan, United States of America
Fredrik Nordvall Forsberg	University of Strathclyde, Scotland) (*chair*)
Elaine Pimentel	University College London, England
Andrew Swan	University of Ljubljana, Slovenia
Tarmo Uustalu	Reykjavík University, Iceland
Niels van der Weide	Radboud University Nijmegen, The Netherlands
Théo Winterhalter	Inria Saclay, France
Maaike Zwart	IT University Copenhagen, Denmark

Steering Committee

Eduardo Hermo Reyes	Formal Vindications, Spain
Tom de Jong	University of Nottingham, England
Rasmus Ejlers Møgelberg	IT University of Copenhagen, Denmark
Fredrik Nordvall Forsberg	University of Strathclyde, Scotland
Paige Randall North	Utrecht University, The Netherlands (*chair*)
Benno van den Berg	University of Amsterdam, The Netherlands (*secretary*)

Local organisers

Guillaume Allais, Stuart Gale, Fredrik Nordvall Forsberg, and the rest of the Mathematically Structured Programming Group at the University of Strathclyde.

Sponsors

Jane Street	(*Gold sponsor*)
Formal Vindications	
Well-Typed	
Input Output	
Tunnock's	

Contents

An alphabetical index of all authors can be found on page 291.

Preface i

Organisation ii

Contents iii

Invited talks 1
- Towards topological type theory for decrypting transfinite methods in classical mathematics . . . 1
- Intuitionistic modalities through proof theory . . . 2
- A Guided Tour of Polarity and Focusing . . . 3
- A constructive higher groupoid model of homotopy type theory . . . 4

Constructive mathematics 5
- Accessible Sets in Martin-Löf Type Theory with Function Extensionality . . . 5
- In Cantor Space, No One Can Hear You Stream . . . 9
- Examples and counter-examples of injective types in univalent mathematics . . . 13

Mechanisation 17
- A Coq Formalization of Lagois Connections for Secure Information Flow . . . 17
- Coq proof of the fifth Busy Beaver value . . . 20
- Mechanizing logical relations . . . 23
- Geometric Reasoning in Lean: from Algebraic Structures to Presheaves . . . 26
- Löb's Theorem and Provability Predicates in Rocq . . . 29

Syntax 34
- Expansion in a Calculus with Explicit Substitutions . . . 34
- Presheaves on Purpose . . . 37
- Thinning Thinnings: Safe and Efficient Binders . . . 40
- Categorical Normalization by Evaluation: A Novel Universal Property for Syntax . . . 43
- Rational Codata as Syntax-with-Binding: Correct-by-Construction Foundations of the Modal μ-Calculus 46

Category theory 49
- Dependent two-sided fibrations for directed type theory . . . 49
- Synthetic-Inductive Category Theory . . . 52
- Directed equality with dinaturality . . . 55
- A Type Theory for Comprehension Categories with Applications to Subtyping . . . 59

Models 62
- Irregular models of type theory . . . 62
- An algebraic internal groupoidal model of Martin-Löf type theory . . . 65
- Realizability Triposes from Sheaves . . . 66
- Solving Guarded Domain Equations in Presheaves Over Ordinals and Mechanizing It . . . 70

Parametricity, and universes 74
- Algebraic Universes and Variances For All . . . 74
- Internalized Parametricity via Lifting Universals . . . 78
- Extending Sort Polymorphism with Elimination Constraints . . . 82
- The Case for Impredicative Universe Polymorphism . . . 86
- From parametricity to identity types . . . 90

Homotopy type theory 93
- About the construction of simplicial and cubical sets in indexed form: the case of degeneracies . . . 93
- Predicative Stone Duality in Univalent Foundations . . . 96
- Cocompleteness in simplicial homotopy type theory . . . 99
- Yet another homotopy group, yet another Brunerie number . . . 102
- Rezk completions For (Elementary) Topoi . . . 106

Philosophy/Logic **110**
Type Theory and Philosophical Logic . . . 110
Constructive algebraic completeness of first-order bi-intuitionistic logic . . . 116

PL applications **120**
Higher-Order Focusing on Linearity and Effects . . . 120
Monadic Equational Reasoning for `while` loop in Rocq . . . 124
Commuting Rules for the Later Modality and Quantifiers in Step-Indexed Logics . . . 127

Equality **131**
How (not) to prove typed type conversion transitive . . . 131
Arrow algebras . . . 134
Weak Equality Reflection in MLTT with Propositional Truncation . . . 137
Towards Quotient Inductive Types in Observational Type Theory . . . 140
Matching (Co)patterns with Cyclic Proofs . . . 144

Quantum and normalisation **147**
Towards a computational quantum logic . . . 147
Internal Proofs of Strong Normalization . . . 151

Compositionality **154**
Compositional memory management in the λ-calculus . . . 154
Choice principles and a cotopological modality in HoTT . . . 158
Towards Modular Composition of Inductive Types Using Lean Meta-programming . . . 161

Dependently Typed Programming **165**
Type-safe Bidirectional Channels in Idris 2 . . . 165
Code Generation via Meta-programming in Dependently Typed Proof Assistants (Experience Report) 170
Towards Being Positively Negative about Dependent Types . . . 174
Implementing a Typechecker for an Esoteric Language: Experiences, Challenges, and Lessons . . . 177
A Two-level Foundation for the Calculus of Constructions . . . 180

Concurrency **183**
A Timed Predicate Temporal Logic Sequent Calculus . . . 183
Verfying Z3 RUP Proofs with the Interactive Theorem Provers Coq/Rocq and Agda . . . 186
Formalising Monitors for Distributed Deadlock Detection . . . 190
Fair Termination for Resource-Aware Active Objects . . . 194
Mechanized safety of Jolteon consensus in Agda . . . 198

Proof theory **201**
Canonical Bidirectional Typing via Polarised System L . . . 201
An existential-free theory of arithmetic in all finite types . . . 204
A Generalized Logical Framework . . . 207
A Curry-Howard correspondence for intuitionistic inquisitive logic . . . 211
Y is not typable in λU . . . 215

Execution **218**
A Ghost Sort for Proof-Relevant yet Erased Data in Rocq and MetaRocq . . . 218
A Fully Dependent Assembly Language . . . 222
Efficient Program Extraction in Elementary Number Theory using the Proof Assistant Minlog . . . 225

Resources **230**
Linear Types inside Dependent Type Theory . . . 230
Impredicative Encodings of Inductive and Coinductive Types . . . 234
NbE for LNL via Adjoint Meta-modalities . . . 237
Unboxed Dependent Types . . . 240
A Quantitative Dependent Type Theory with Recursion . . . 243

Containers **246**
Stellar — A Library for API Programming . . . 246
An Inductive Universe for Setoids . . . 250
Weihrauch Problems as Containers . . . 253
Containers: Compositionality for Tensors . . . 257
Large Elimination and Indexed Types in Refinement Types . . . 260

Mechanised meta theory **263**
HoTTLean: Formalizing the Meta-Theory of HoTT in Lean . . . 263

Towards Formalising the Guard Checker of Rocq . . . 267
Lean4Lean: Mechanizing the Metatheory of Lean . . . 271
AdapTT: A Type Theory with Functorial Types . . . 274

Models (second session) **278**
Cohomology in Synthetic Stone Duality . . . 278
Representing type theories in two-level type theory . . . 281
Extensional concepts in intensional type theory, revisited . . . 284
Lightweight Agda Formalization of Denotational Semantics . . . 286

Author index **291**

Towards topological type theory for decrypting transfinite methods in classical mathematics

Ingo Blechschmidt

University of Antwerp

In constructive mathematics, an ongoing challenge is to extracting computational content from infinitary arguments in classical mathematics — especially those that yield concrete results via abstract tools such as minimal bad sequences or maximal ideals. Today, a wide range of such techniques exists, constituting a late partial fulfillment of Hilbert's program.

This talk presents work in progress on porting the modal operators "everywhere" and "somewhere", as developed in set theory by Joel David Hamkins, Victoria Gitman and their collaborators, into type theory. This modal enrichment provides a uniform language for expressing several established constructivization techniques, and is at the core of a relatively recent new technique that offers insights on the origins of some of our most cherished inductive definitions. These developments are closely connected to recent advances in type-theoretic presheaf and sheaf models, dialogue continuity, synthetic algebraic geometry and well-quasi-order theory.

Intuitionistic modalities through proof theory

Sonia Marin

University of Birmingham
s.marin@bham.ac.uk

Intuitionistic modal logic, despite more than seventy years of investigation, [5], still partly escapes our comprehension. Structural proof theoretic accounts of intuitionistic modal logic have adopted either the paradigm of *labelled deduction* in the form of labelled natural deduction and sequent systems [7], or the one of *unlabelled deduction* in the form of sequent [2] or nested sequent systems [8, 1]. Both of these approaches are still under active consideration. In this talk, I would like to give an overview of the current landscape of intuitionistic modal proof theory and survey some of the more recent developments [3, 4, 6].

References

1. Arisaka, R., Das, A., Strasßburger, L.: On nested sequents for constructive modal logics. Logical Methods in Computer Science **11**(3) (2015)
2. Bierman, G.M., de Paiva, V.: Intuitionistic necessity revisited. Tech. rep., School of Computer Sciences-University of Birmingham (1996)
3. Das, A., van der Giessen, I., Marin, S.: Intuitionistic Gödel-Löb logic, à la Simpson: Labelled systems and birelational semantics. In: Murano, A., Silva, A. (eds.) CSL 2024, Proceedings (2024)
4. Das, A., Marin, S.: On intuitionistic diamonds (and lack thereof). In: Ramanayake, R., Urban, J. (eds.) TABLEAUX 2023 (2023)
5. Fitch, F.B.: Intuitionistic modal logic with quantifiers **7**(2), 113–118 (1948)
6. Girlando, M., Kuznets, R., Marin, S., Morales, M., Straßburger, L.: A simple loopcheck for intuitionistic K. In: Metcalfe, G., Studer, T., de Queiroz, R.J.G.B. (eds.) WoLLIC 2024, Proceedings (2024)
7. Simpson, A.: The Proof Theory and Semantics of Intuitionistic Modal Logic. Ph.D. thesis, University of Edinburgh (1994)
8. Straßburger, L.: Cut elimination in nested sequents for intuitionistic modal logics. In: International Conference on Foundations of Software Science and Computational Structures. pp. 209–224. Springer (2013)

A Guided Tour of Polarity and Focusing

Chris Martens

Northeastern University

Focusing and polarity are two intertwined ideas that relate proofs and computation, informing modern ideas about canonical forms, pattern matching, evaluation order, logical frameworks, and logic programming in the proof-search-as-computation paradigm. However, the relationship between these ideas, their history, and the underlying proof theory, remain obscure and folkloric to many practitioners. This talk will serve as beginner's guide for the uninitiated, attempting to reconcile disparate presentations, separate orthogonal concepts, signpost the landscape of important prior work in the area, and gesture at promising paths for further investigation.

A constructive higher groupoid model of homotopy type theory

Christian Sattler

Chalmers University of Technology and University of Gothenburg

In homotopy type theory, every type effectively carries the structure of a *higher groupoid*: a set equipped with a higher-dimensional equivalence relation. Conversely, there ought to be an effective interpretation of homotopy type theory in higher groupoids. Unfortunately, Voevodsky's model in Kan simplicial sets is non-constructive. And existing cubical models fail to constructively present higher groupoids in the above sense.

In this talk, I will introduce an effective interpretation of homotopy type theory in higher groupoids. This is based on a combination of cubical and semisimplicial techniques.

Accessible Sets in Martin-Löf Type Theory with Function Extensionality

Yuta Takahashi

Aomori University, Aomori, Japan
y.takahashi@aomori-u.ac.jp

Background: Accessible Sets in Cumulative Hierarchy After Bishop [5] showed that analysis can be developed constructively without using Brouwer's intuitionistic mathematics, several authors investigated a system to formalise Bishop's constructive analysis. For instance, Myhill [13] introduced a system of constructive set theory, and later Aczel [1, 2, 3] introduced another system of constructive set theory called constructive Zermelo-Fraenkel set theory **CZF**. On the other hand, Martin-Löf [12] took an approach different from the set-theoretic one: he formulated a framework of constructive type theory called Martin-Löf type theory **MLTT**. This framework follows the Curry-Howard correspondence, and at the same time, comprises a set of rules to define mathematical objects inductively or recursively.

Aczel's work on **CZF** also showed that these two approaches are compatible. He defined a cumulative hierarchy $\mathbb{V}$ of sets as a W-type in **MLTT**, and interpreted all axioms of **CZF** in **MLTT**. This hierarchy can be considered as a setoid model of **CZF**: $\mathbb{V}$ is a type with the equivalence relation $\doteq$, which is defined by the induction principle on $\mathbb{V}$ as a W-type. For any set $a : \mathbb{V}$, one can also define the type $\mathsf{index}\, a$ and the set $\mathsf{pred}\, a\, x$, which are the type of indices for the elements of a and the element of a of index x, respectively. We abbreviate $(i : \mathsf{index}\, a) \to \Phi(\mathsf{pred}\, a\, i)$ as $\forall_{(x \in a)} \Phi(x)$.

As in classical Zermelo-Fraenkel set theory, the transitive closure of a set can be defined in **CZF** (see, *e.g.*, [9]). Recall that the transitive closure $\mathbf{TC}(a)$ of a set a satisfies the equation

$$\mathbf{TC}(a) = a \cup \bigcup \{\mathbf{TC}(x) \mid x \in a\},$$

which implies that $\mathbf{TC}(a)$ is a transitive set:

$$\forall x \forall y (y \in x \in \mathbf{TC}(a) \to y \in \mathbf{TC}(a)).$$

So the transitive closure of a set a contains all sets below a in the hierarchy as its elements. Through Aczel's interpretation of **CZF**, one has the corresponding operator $\mathsf{tc} : \mathbb{V} \to \mathbb{V}$ in **MLTT**.

By using Dybjer's indexed inductive definition [7], one can then define the *accessibility* $\mathsf{Acc} : \mathbb{V} \to \mathsf{Set}$ with the constructor prog, the eliminator $\mathsf{ind}_{\mathsf{Acc}}$, and the computation rule below: for any universe level ℓ,

$$\begin{aligned}
\mathsf{prog} &: (a : \mathbb{V}) \to \forall_{(x \in \mathsf{tc}\, a)} \mathsf{Acc}\, x \to \mathsf{Acc}\, a \\
\mathsf{ind}_{\mathsf{Acc}} &: (P : (a : \mathbb{V}) \to \mathsf{Acc}\, a \to \mathsf{Set}\, \ell) \to \\
&\qquad ((a : \mathbb{V})(f : \forall_{(x \in \mathsf{tc}\, a)} \mathsf{Acc}\, x) \to \\
&\qquad\quad ((i : \mathsf{index}\, (\mathsf{tc}\, a)) \to P\, (\mathsf{pred}\, (\mathsf{tc}\, a)\, i)\, (f\, i)) \to P\, a\, (\mathsf{prog}\, a\, f)) \to \\
&\qquad (a : \mathbb{V})(c : \mathsf{Acc}\, a) \to P\, a\, c \\
\mathsf{ind}_{\mathsf{Acc}}\, P\, h\, a\, (\mathsf{prog}\, a\, f) &= h\, a\, f\, (\lambda i.\mathsf{ind}_{\mathsf{Acc}}\, P\, h\, (\mathsf{pred}\, (\mathsf{tc}\, a)\, i)\, (f\, i))
\end{aligned}$$

Roughly speaking, $\mathsf{Acc}\,a$ means that the set a is constructed from below: the constructor prog says that if all sets below a are constructed then a is constructed as well. The eliminator $\mathsf{ind}_{\mathsf{Acc}}$ together with the computation rule provides the induction principle along such construction. Note that Acc is a special case of the *accessible part* of a binary relation, which can be defined as an inductive family by Dybjer's indexed inductive definition. The notion of accessible part has several applications in the areas of research such as proof theory and term rewriting (see, *e.g.*, [14] and [4], respectively).

The induction principle for Acc is stronger than the W-induction principle on $a : \mathbb{V}$ in the sense that the former admits the induction hypothesis not only for an arbitrary $x \in a$, but also for an arbitrary member of the transitive closure $\mathsf{tc}\,a$. For instance, let

$$\mathsf{nextU} : \Sigma_{(A:\mathsf{Set})}(A \to \mathsf{Set}) \to \Sigma_{(A:\mathsf{Set})}(A \to \mathsf{Set})$$

be a universe operator which takes a family of types and returns a Tarski-universe (U, T) containing all types in this family. Using the induction principle for Acc, one can define a hierarchy $\mathsf{U}\,a\,t$ of Tarski-universes with $a : \mathbb{V}$ and $t : \mathsf{Acc}$ as follows.

$$\mathsf{U}\,a\,(\mathsf{prog}\,a\,f) = \mathsf{fst}\left(\mathsf{nextU}(\mathsf{index}\,(\mathsf{tc}\,a), \lambda i.\mathsf{U}\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,i)\,(f\,i))\right)$$

Roughly speaking, $\mathsf{U}\,a\,(\mathsf{prog}\,a\,f)$ contains as its subuniverses not only $\mathsf{U}\,v\,t$ for any $v \in a$ with $t : \mathsf{Acc}\,v$, but also $\mathsf{U}\,w\,s$ for any $w \in \mathsf{tc}\,a$ with $s : \mathsf{Acc}\,w$.

Though the transitive closure operator tc is accompanied by a similar induction principle which is stronger than the W-induction principle on $\mathbb{V}$, the Acc-induction principle has an advantage over that of tc: the Acc-induction principle has the simple and useful computation rule as seen above. One might try to verify that the tc-induction principle would have the following computation rule

$$\mathsf{ind}_{\mathsf{tc}}\,P\,h\,a = h\,a\,(\lambda i.\mathsf{ind}_{\mathsf{tc}}\,P\,h\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,i))$$

for any $a : \mathbb{V}$ with $P : \mathbb{V} \to \mathsf{Set}\,\ell$ and $h : (a : \mathbb{V}) \to \forall_{(x \in \mathsf{tc}\,a)} P\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,x) \to P\,a$. However, in intensional **MLTT** it is implausible that one can obtain this computation rule, and in fact one should not obtain, otherwise the evaluation of $\mathsf{ind}_{\mathsf{tc}}\,P\,h\,a$ by the ξ-rule does not terminate:

$$\begin{aligned}\mathsf{ind}_{\mathsf{tc}}\,P\,h\,a &= h\,a\,(\lambda i.\mathsf{ind}_{\mathsf{tc}}\,P\,h\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,i)) \\ &= h\,a\,(\lambda i.h\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,i)(\lambda j.\mathsf{ind}_{\mathsf{tc}}\,P\,h\,(\mathsf{pred}\,(\mathsf{tc}\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,i))\,j))) = \cdots\end{aligned}$$

Aim: Accessible Sets under Function Extensionality We show that the constructor and eliminator of Acc, namely, the introduction and elimination rules of Acc are derivable in **MLTT** with function extensionality: for any universe levels ℓ_1 and ℓ_2,

$$\begin{aligned}&\mathsf{funext} : (A : \mathsf{Set}\,\ell_1)(B : A \to \mathsf{Set}\,\ell_2) \\ &\qquad (f\,g : (x : A) \to B\,x) \to ((x : A) \to f\,x =_{B\,x} g\,x) \to f =_{(x:A)\to B\,x} g.\end{aligned}$$

This means that indexed inductive definition is dispensable for formulating the constructor and eliminator of Acc in **MLTT** with function extensionality, though its computation rule is still missing without indexed inductive definition.

For this purpose, we first derive the induction principle on transitive closure tc without using function extensionality: for any universe level ℓ,

$$\mathsf{ind}_{\mathsf{tc}} : (P : \mathbb{V} \to \mathsf{Set}\,\ell) \to ((a : \mathbb{V}) \to \forall_{(x \in \mathsf{tc}\,a)} P\,x \to P\,a) \to (a : \mathbb{V}) \to P\,a.$$

The accessibility $\mathsf{Acc} : \mathbb{V} \to \mathsf{Set}$ is then defined as follows: putting $P := \lambda a.\mathsf{Set}$, we define

$$\mathsf{acc} := \lambda a.\lambda g.(i : \mathsf{index}\,(\mathsf{tc}\,a)) \to g\,i, \qquad \mathsf{Acc} := \mathsf{ind}_{\mathsf{tc}}\,P\,\mathsf{acc}.$$

Next, using function extensionality, we prove the following *propositional* computation rule of $\mathsf{ind}_{\mathsf{tc}}$. Essentially, this rule was already proved in the present author's preprint [15, Appendix].

Proposition. *For any* $P : \mathbb{V} \to \mathsf{Set}\,\ell$, $h : (a' : \mathbb{V}) \to \forall_{(x \in \mathsf{tc}\,a')} P\,x \to P\,a'$ *and* $a : \mathbb{V}$, *we have the following term of the identity type* $\mathsf{ind}_{\mathsf{tc}}\,P\,h\,a =_{P\,a} h\,a\,(\lambda i.\mathsf{ind}_{\mathsf{tc}}\,P\,h\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,i))$.

$$\mathsf{comp}_{\mathsf{tc}}\,P\,h\,a : \mathsf{ind}_{\mathsf{tc}}\,P\,h\,a =_{P\,a} h\,a\,(\lambda i.\mathsf{ind}_{\mathsf{tc}}\,P\,h\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,i)).$$

If one takes P and acc as above, then an instance

$$\mathsf{comp}_{\mathsf{tc}}\,P\,\mathsf{acc}\,a : \mathsf{Acc}\,a =_{\mathsf{Set}} \forall_{(x \in \mathsf{tc}\,a)}\mathsf{Acc}\,x$$

is obtained, which is crucial to our argument. Since the direction from the right of this identity type to the left corresponds to the introduction rule of Acc, we can define the constructor $\mathsf{prog} : (a : \mathbb{V}) \to \forall_{(x \in \mathsf{tc}\,a)}\mathsf{Acc}\,x \to \mathsf{Acc}\,a$ by transporting along this direction:

$$\mathsf{prog} := \lambda a.\lambda f.\mathsf{transport}\,(\lambda A.A)\,(\mathsf{sym}\,(\mathsf{comp}_{\mathsf{tc}}\,P\,\mathsf{acc}\,a))\,f.$$

To derive the Acc-elimination rule, we prove $(a : \mathbb{V})(c : \mathsf{Acc}\,a) \to P\,a\,c$ under the assumptions of this rule by induction on transitive closure of a. Transporting from $\mathsf{Acc}\,a'$ to $\forall_{(x \in \mathsf{tc}\,a')}\mathsf{Acc}\,x$ for any $a' : \mathbb{V}$ provides the function $\mathsf{inv} : (a' : \mathbb{V}) \to \mathsf{Acc}\,a' \to \forall_{(x \in \mathsf{tc}\,a')}\mathsf{Acc}\,x$ as

$$\mathsf{inv} := \lambda a'.\lambda c'.\mathsf{transport}\,(\lambda A.A)\,(\mathsf{comp}_{\mathsf{tc}}\,P\,\mathsf{acc}\,a')\,c',$$

so we have $\mathsf{inv}\,a\,c : \forall_{(x \in \mathsf{tc}\,a)}\mathsf{Acc}\,x$. By using the assumption

$$(a : \mathbb{V})(f : \forall_{(x \in \mathsf{tc}\,a)}\mathsf{Acc}\,x) \to ((i : \mathsf{index}\,(\mathsf{tc}\,a)) \to P\,(\mathsf{pred}\,(\mathsf{tc}\,a)\,i)\,(f\,i)) \to P\,a\,(\mathsf{prog}\,a\,f)$$

with the induction hypothesis $\forall_{(x \in \mathsf{tc}\,a)}(d : \mathsf{Acc}\,x) \to P\,x\,d$, we have $P\,a\,(\mathsf{prog}\,a\,(\mathsf{inv}\,a\,c))$. As a general fact on $\mathsf{transport}$, we also have

$$\begin{aligned}(A : \mathsf{Set}\,\ell_1)(P : A \to \mathsf{Set}\,\ell_2)(x\,y : A)(p : x =_A y)(c : P\,x)\\ \to \mathsf{transport}\,P\,(\mathsf{sym}\,p)\,(\mathsf{transport}\,P\,p\,c) =_{P\,x} c,\end{aligned}$$

so $\mathsf{prog}\,a\,(\mathsf{inv}\,a\,c) =_{\mathsf{Acc}\,a} c$ holds. Hence it follows from $P\,a\,(\mathsf{prog}\,a\,(\mathsf{inv}\,a\,c))$ that $P\,a\,c$ holds.

In addition, we show with function extensionality that for any $a : \mathbb{V}$ there is a unique proof of $\mathsf{Acc}\,a$, that is,

$$(a : \mathbb{V}) \to \Sigma_{(t : \mathsf{Acc}\,a)}((s : \mathsf{Acc}\,a) \to t =_{\mathsf{Acc}\,a} s)$$

holds. We formalised our result in Agda by postulating function extensionality [16].

Future Work Aczel's interpretation of **CZF** in **MLTT** was refined in Homotopy type theory (HoTT) [17]. The cumulative hierarchy $\mathbb{V}$ of sets is defined not as a W-type but as a higher inductive type, where the equivalence relation $\doteq$ on $\mathbb{V}$ is replaced with the identity type $=_{\mathbb{V}}$ and $\mathbb{V}$ has the path constructor for $=_{\mathbb{V}}$. Other interpretations of **CZF** in HoTT were investigated in, *e.g.*, [10, 11, 8]. In the literature of HoTT the accessibility in general, namely, the accessible part of a binary relation is defined by indexed inductive definition [17, 6]. A future research direction is to examine whether its special case Acc is derivable in HoTT by adapting our argument above: we will examine whether the constructor and eliminator of Acc are derivable under some interpretation of **CZF** in HoTT, which can prove function extensionality by means of the univalence axiom.

References

[1] Peter Aczel. The type theoretic interpretation of constructive set theory. In Angus Macintyre, Leszek Pacholski, and Jeff Paris, editors, *Logic Colloquium '77*, volume 96 of *Studies in Logic and the Foundations of Mathematics*, pages 55–66. Elsevier, 1978.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory: Choice principles. In A. S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, pages 1–40. North-Holland, 1982.

[3] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definitions. In R. B. Marcus, G. J. Dorn, and G. J. W. Dorn, editors, *Logic, Methodology, and Philosophy of Science VII*, pages 17–49. North-Holland, 1986.

[4] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, 1967.

[6] Tom de Jong, Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. Set-theoretic and type-theoretic ordinals coincide. In *LICS*, pages 1–13, 2023.

[7] Peter Dybjer. Inductive families. *Formal Aspects Comput.*, 6(4):440–465, 1994.

[8] Cesare Gallozzi. Homotopy type-theoretic interpretations of constructive set theories. *Math. Struct. Comput. Sci.*, 31(1):112–143, 2021.

[9] Edward R. Griffor and Michael Rathjen. The strength of some Martin-Löf type theories. *Arch. Math. Log.*, 33(5):347–385, 1994.

[10] Håkon Robbestad Gylterud. From multisets to sets in homotopy type theory. *J. Symb. Log.*, 83(3):1132–1146, 2018.

[11] Håkon Robbestad Gylterud. Multisets in type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):1–18, 2020.

[12] Per Martin-Löf. An intuitionistic theory of types. In G. Sambin and Jan M. Smith, editors, *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*, pages 127–172. Clarendon Press, 1998.

[13] John Myhill. Constructive set theory. *The Journal of Symbolic Logic*, 40(3):347–382, 1975.

[14] Wolfram Pohlers. *Proof Theory: The First Step into Impredicativity*. Springer, Berlin, Heidelberg, 2009.

[15] Yuta Takahashi. Interpretation of Inaccessible Sets in Martin-Löf Type Theory with One Mahlo Universe. *CoRR*, abs/2402.15074, 2024.

[16] Yuta Takahashi. Accessibility, 2025. https://github.com/takahashi-yt/accessibility.

[17] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

In Cantor Space, No One Can Hear You Stream

Martin Baillon, Assia Mahboubi, and Pierre-Marie Pédrot

INRIA, France

It is well-known that all computable functions are continuous. This result has been enshrined in various kinds of Brouwerian intuitionistic mathematics through flavours of bar recursion [19] with varying degrees of compatibility with classical mathematics. In type theory, continuity is usually phrased as a property of functions $\varphi : (\mathbb{N} \to A) \to B$ for suitable types A and B, and boils down to the fact that such a function can only depend on a finite prefix of any argument $\alpha : \mathbb{N} \to A$. In this abstract, we will care about the case $A := \mathbb{B}$, for which $\mathbb{N} \to \mathbb{B}$ is known as the *Cantor space*, and B is some hereditarily positive type, i.e. purely first-order inductive, typically $B := \mathbb{N}$.

Streams of booleans are incomprehensible beings. Infinite processions crawling around the stars like never-ending snakes, they stay in the shadows, lurking in the outskirts of computation. For cardinality reasons, most of them cannot even be named. Ungraspable, continuity of all functionals over the Cantor space means that they will forever stay alien to our human understanding. No function can actually witness their horror unheeded. In Cantor space, no one can hear you stream.

As anecdotal as it may seem, continuity and its alter-egos have extremely strong consequences, at least when given internally. For instance, bar recursion is enough to realize the principle known as double-negation shift, and thus to reach the logical strength of second-order arithmetic in a system that is otherwise simply-typed [5]. This is one of the reasons for the importance of such principles in the eyes of the Brouwerian crowd, who used them to develop constructive analysis. Continuity is furthermore tightly related to choice principles [6].

There has been a revival around this topic in recent years. In addition to the already mentioned papers, Escardó gave a proof that System T enjoys continuity in an external way [13] using a kind of side-effect reminiscent of interaction-trees [20]. Baillon et al. gave a generalization of this proof to Baclofen Type Theory, a variant of MLTT with a restricted form of large elimination [4]. Escardó and Xu also studied more semantical approaches to the same kinds of questions [21, 12]. From the community of PER semantics, there was an important series of papers by several authors including Rahli in the intersection of all of them [18, 16, 17, 7, 8, 9] considering various forms of continuity in realizability models of MLTT.

The work we describe here can be seen as a variant on the models considered in the latter trend. It lies in the continuity of the first author's PhD thesis [3] and can be seen as a mechanized refinement of a previous work by Coquand and Jaber [11]. We prove external continuity of functionals over the Cantor space in MLTT, i.e. terms of type $\vdash M : (\mathbb{N} \to \mathbb{B}) \to \mathbb{N}$ by embedding them into a larger type theory MLTT^F which is basically MLTT where terms are extended with a single abstract oracle $\alpha : \mathbb{N} \to \mathbb{B}$ and contexts with partial knowledge about the oracle

$$\Gamma := \ldots \mid \Gamma, \alpha(n) \mapsto b \qquad n \in \mathbf{N}, b \in \mathbf{B}$$

where $\mathbf{N} := \{0, 1, \ldots\}$ and $\mathbf{B} := \{\mathtt{tt}, \mathtt{ff}\}$ stand respectively for integer and boolean literals. Such a hypothesis intuitively means that the value of α at n is known to be b. This partial knowledge can be reflected equationally through the conversion rule below.

$$\frac{\alpha(n) \mapsto b \in \Gamma}{\Gamma \vdash \alpha\ \overline{n} \equiv \overline{b} : \mathbb{B}}$$

Moreover, all kinds of judgements of $\mathsf{MLTT}^{\digamma}$ are extended with a built-in *splitting* principle that allows accumulating knowledge about the oracle. For instance, for term typing we have

$$\frac{\Gamma, \alpha(n) \mapsto \mathtt{tt} \vdash M : A \qquad \Gamma, \alpha(n) \mapsto \mathtt{ff} \vdash M : A \qquad \alpha(n) \mapsto - \notin \Gamma}{\Gamma \vdash M : A}$$

It is somewhat immediate to the astute reader that the intended model for this theory is some kind of sheaf model [15], i.e. a proof-relevant version of Beth semantics. Basing ourselves upon the `logrel-coq` development [2], we prove in Coq that $\mathsf{MLTT}^{\digamma}$ enjoys a form of canonicity and strong normalization using a logical relation model similar to the now famous NbE model of Abel et al. [1]. While our syntax is essentially the same as the one from the Coquand-Jaber note [11], we prove a somewhat more general result. Our model is indeed a kind of realizability, i.e. based on reduction instead of evaluation in the metatheory. As a result, rather than just equational canonicity, we also prove that well-typed terms actually *reduce* to a normal form. As is usual in sheaf models, canonicity is weakened to some kind of decision tree, e.g. for $\mathbb{N}$ it implies that for every closed $\vdash M : \mathbb{N}$ there is a finite splitting tree such that M reduces to an integer on each branch. This is a generalization of the trivial case of canonicity under consistent, purely negative axioms [10].

Furthermore, our model features a first-class handling of variables, so it scales to open terms. This is achieved by the standard trick ensuring that all relations are presheaves over contexts and that *neutrals* are always realizers. While this technique is well-known, the resulting property offers a stark contrast with the models of Rahli et al., such as $\mathrm{TT}^{\square}_{\mathcal{C}}$ [9], where only *closed* terms are manipulated in the semantics. The latter hardwires some logical consequences in the internal language, such as equality reflection and, in particular, function extensionality. Unfortunately, apart from not being provable in MLTT and making type-checking undecidable, extensionality is often at odds with strong continuity principles. This forces some of the strongest continuity principles to be made innocuous by squashing them, thus rendering them computationally irrelevant.

Just like a proper handling of variables makes the notion of neutrals paramount, our work highlights the importance of their sheaf equivalent, that we call *α-neutrals*. They are basically the equivalent of neutrals, where the head is not a variable but $\alpha\ n$ with $\alpha(n) \mapsto - \notin \Gamma$. Like mere neutrals, they block computation but contrary to them, an α-neutral can be unlocked by a weakening adding $\alpha(n) \mapsto b$. Indeed, extending contexts can increase computational knowledge about the oracle. The resulting logical relation also smells very strongly of call-by-push-value (CBPV) [14] and more generally of semantics of effectful languages. As a matter of fact, the relation is split between a strong and a weak variant. The former corresponds to pure head values that are not allowed to split, while the latter corresponds to effectful terms that require splitting to be well-typed.

We will give insights about this model and describe some of the ongoing work on the formalization and the expected results to come.

References

[1] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158111.

[2] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2024, page 230–245, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3636501.3636951.

[3] Martin Baillon. *Continuity in Type Theory*. Theses, Nantes Université, December 2023. URL: https://theses.hal.science/tel-04617881.

[4] Martin Baillon, Assia Mahboubi, and Pierre-Marie Pédrot. Gardening with the pythia A model of continuity in a dependent setting. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CSL.2022.5.

[5] Valentin Blot. A direct computational interpretation of second-order arithmetic via update recursion. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 62:1–62:11. ACM, 2022. doi:10.1145/3531130.3532458.

[6] Nuria Brede and Hugo Herbelin. On the logical structure of choice and bar induction principles. In *LICS*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470523.

[7] Liron Cohen, Bruno da Rocha Paiva, Vincent Rahli, and Ayberk Tosun. Inductive continuity via Brouwer trees. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*, volume 272 of *LIPIcs*, pages 37:1–37:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.MFCS.2023.37.

[8] Liron Cohen and Vincent Rahli. Realizing continuity using stateful computations. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13-16, 2023, Warsaw, Poland*, volume 252 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CSL.2023.15.

[9] Liron Cohen and Vincent Rahli. $\mathrm{TT}^{\Box}_{\mathcal{C}}$: A family of extensional type theories with effectful realizers of continuity. *Logical Methods in Computer Science*, Volume 20, Issue 2, Jun 2024. doi:10.46298/lmcs-20(2:18)2024.

[10] Thierry Coquand, Nils Anders Danielsson, Martín Hötzel Escardó, Ulf Norell, and Chuangjie Xu. Negative consistent axioms can be postulated without loss of canonicity. 2013. URL: https://martinescardo.github.io/papers/negative-axioms.pdf.

[11] Thierry Coquand and Guilhem Jaber. A note on forcing and type theory. *Fundam. Inf.*, 100(1–4):43–52, January 2010.

[12] Martín Escardó and Chuangjie Xu. A constructive manifestation of the Kleene-Kreisel continuous functionals. *Ann. Pure Appl. Log.*, 167(9):770–793, 2016. doi:10.1016/J.APAL.2016.04.011.

[13] Martín Hötzel Escardó. Continuity of Gödel's system T definable functionals via effectful forcing. In *MFPS*, volume 298 of *Electronic Notes in Theoretical Computer Science*, pages 119–141. Elsevier, 2013. doi:10.1016/j.entcs.2013.09.010.

[14] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Kluwer Academic Publishers, USA, 2004.

[15] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic*. Springer New York, 1992.

[16] Vincent Rahli and Mark Bickford. Validating Brouwer's continuity principle for numbers using named exceptions. *Math. Struct. Comput. Sci.*, 28(6):942–990, 2018. doi:10.1017/S0960129517000172.

[17] Vincent Rahli, Mark Bickford, Liron Cohen, and Robert L. Constable. Bar induction is compatible

with constructive type theory. *J. ACM*, 66(2):13:1–13:35, 2019. doi:10.1145/3305261.

[18] Vincent Rahli, Mark Bickford, and Robert L. Constable. Bar induction: The good, the bad, and the ugly. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005074.

[19] Anne Sjerp Troelstra and Dirk van Dalen. Constructivism in mathematics. vol. I. *Studies in Logic and the Foundations of Mathematics*, 26, 1988.

[20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.

[21] Chuangjie Xu and Martín Hötzel Escardó. A constructive model of uniform continuity. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, volume 7941 of *Lecture Notes in Computer Science*, pages 236–249. Springer, 2013. doi:10.1007/978-3-642-38946-7_18.

Examples and counter-examples of injective types in univalent mathematics

Tom de Jong[1] and Martín Hötzel Escardó[2]

[1] University of Nottingham, Nottingham, UK
tom.dejong@nottingham.ac.uk
[2] University of Birmingham, Birmingham, UK
m.escardo@bham.ac.uk

In the context of univalent foundations [17], injective types were introduced by the second author in [6]. The interest in injectivity originated in its use to construct infinite searchable types [5], but the topic turned out to have a rather rich theory on its own. We present new examples and counter-examples of injective types from our Agda development [7].

Definition 1. A type D is (algebraically) **injective** if it has the property that for every embedding $j : X \hookrightarrow Y$, any map $f : X \to D$ into D has a designated extension f/j. By extension, we mean that the diagram below commutes, i.e. $f/j \circ j = f$ holds.

$$\begin{array}{ccc} X & \xhookrightarrow{\ j\ } & Y \\ & {\scriptstyle f}\searrow \quad \swarrow{\scriptstyle f/j} & \\ & D & \end{array} \tag{$\dagger$}$$

We recall that an embedding is a map whose fibers are propositions. The algebraicity here refers to the fact that we ask for a specified extension, i.e. formulated with a Σ-type, rather than the existence of some unspecified extension, i.e. formulated with $\exists$, the propositional truncation of a Σ-type. Here we consider only algebraically injective types and abuse terminology by dropping the adjective.

With classical logic, i.e. in the presence of excluded middle, the injective types are precisely the pointed types. In fact, this characterization is equivalent to excluded middle. In constructive univalent foundations, the situation is more interesting, as we show in this abstract.

Injectivity and universe levels. The notion of injectivity is universe dependent [6], in the absence of propositional resizing, so that we are led, for universes $\mathcal{U}$, $\mathcal{V}$ and $\mathcal{W}$, to consider types $D : \mathcal{W}$ that are $(\mathcal{U}, \mathcal{V})$-injective in the sense that the types X and Y in ($\dagger$) are restricted to live in $\mathcal{U}$ and $\mathcal{V}$, respectively. The following theorem says that there are no non-trivial *small* injective types in general,[1] and is comparable to Corollary 10 of [1] which says that in the predicative set theory CZF it is consistent that the only injective *sets* (as opposed to classes) are singletons.

Theorem 2. If there is a $(\mathcal{U},\mathcal{U})$-injective type in $\mathcal{U}$ with two distinct points, then the type $\Omega_{\neg\neg} :\equiv \Sigma\, P : \mathcal{U}, \text{is-prop}(P) \times (\neg\neg P \to P)$ of $\neg\neg$-stable propositions in $\mathcal{U}$, whose native universe is $\mathcal{U}^+$, is equivalent to a type in $\mathcal{U}$.

For simplicity, we will not pay too much attention to the universe levels here, but we stress that they are important and that our Agda development [7] in `TypeTopology` rigorously keeps track of them.

[1] The conclusion of Theorem 2, the resizing of $\Omega_{\neg\neg}$, is not provable in univalent foundations, as observed by Andrew Swan. Given a small copy of $\Omega_{\neg\neg}$, we can interpret classical second order arithmetic via $\neg\neg$-stable propositions and subsets, but the consistency strength of univalent foundations is below that of classical second order arithmetic by [14, Corollary 6.7].

Examples. The following is a non-exhaustive list of examples of injective types and extends [6]:

1. Any *univalent* type universe $\mathcal{U}$. Indeed, given $j : X \hookrightarrow Y$ and a type family $f : X \to \mathcal{U}$, we can define $(f/j)\, y :\equiv \Sigma\,(x, -) : j^{-1}(y), f\, x$ where $j^{-1}(y) :\equiv \Sigma\, x : X, f\, x = y$ denotes the fiber of j at y. We note that defining an extension using Π instead of Σ also works.
2. The type of propositions $\Omega_{\mathcal{U}}$ in $\mathcal{U}$. Similar to above, we have extensions via Π and Σ.
3. The type of ordinals in $\mathcal{U}$, with extensions given by suprema [2, Thm. 5.8] for instance.
4. The type of iterative (multi)sets [9, 10] in $\mathcal{U}$.
5. The types of (small) ∞-magmas, monoids, and groups.
6. The type $\mathcal{L}\, X :\equiv \Sigma\, P : \Omega_{\mathcal{U}}, (P \to X)$ of partial elements [4] of any type $X : \mathcal{U}$.
7. The underlying type of any sup-complete poset, and more generally, of any pointed dcpo.

Examples 1, 2 and 6 were already present in [6], as is the fact that injective types are closed under retracts and dependent products. We now also have a sufficient criterion for a Σ-type over an injective type to be injective which accounts for the examples of Item 5. For *subtypes* of injective types there is a necessary and sufficient condition:

Theorem 3. A subtype $\Sigma\, d : D, P\, d$ of an injective type D is injective if and only if we have $f : D \to D$ such that for all $d : D$, the property P holds for $f\, d$, and $P\, d$ implies $f\, d = d$.

In particular, any reflective subuniverse [16] is injective, which also follows from [6, Theorem 24].

Counter-examples. The only type that is *provably* not injective is the empty type, because classically any pointed type is injective. But there are plenty of examples of types that cannot be shown to be injective in constructive mathematics, because their injectivity would imply a *constructive taboo*: a statement that is not constructively provable and is false in some models.

The relevant taboo in this case is **weak excluded middle** which says that for any proposition P either $\neg P$ or $\neg\neg P$ holds, and which is equivalent to De Morgan's law [11, Prop. D4.6.2].

Theorem 4. If any of the following types is injective, then weak excluded middle holds.

1. The type of booleans $\mathbf{2} :\equiv \mathbf{1} + \mathbf{1}$. (This counter-example already appears in *loc. cit.*)
2. The simple types, obtained from $\mathbb{N}$ by iterating function types.
3. The type of Dedekind reals.
4. The type of conatural [3] numbers $\mathbb{N}_\infty :\equiv \Sigma\, \alpha : \mathbb{N} \to \mathbf{2}, (\Pi\, i : \mathbb{N}, \alpha_i \geq \alpha_{i+1})$.
5. More generally, any type with an apartness relation and two points apart.

Counter-example 5 implies that none of the examples of injective types given above have interesting apartness relations. In particular, this result may be seen as an internal version of Kocsis' result [12, Corollary 5.7] that MLTT does not define any non-trivial apartness relation on a universe (in *loc. cit.* this fact is obtained by a parametricity argument).

In computation, it is important to identify decidable properties of types. The following Rice-like [15] theorem says that injective types have no non-trivial decidable properties.

Theorem 5. If an injective type has a decomposition, then weak excluded middle holds.

Here, a *decomposition* of a type X is defined to be a function $f : X \to \mathbf{2}$ such that we have $x_0 : X$ and $x_1 : X$ with $f\, x_0 = 0$ and $f\, x_1 = 1$.

While the type $\Sigma\, X : \mathcal{U}, X$ of pointed types and the type $\Sigma\, X : \mathcal{U}, \neg\neg X$ of non-empty types are both injective, the type of inhabited types is not in general.

Proposition 6. The type $\Sigma\, X : \mathcal{U}, \|X\|$ of inhabited types is injective if and only if *all propositions are projective* [13] (a weak choice principle that fails in some toposes [8]).

The above illustrates the constructive difference between the double negation and the propositional truncation (which coincide if and only if excluded middle holds).

References

[1] Peter Aczel, Benno van den Berg, Johan Granström, and Peter Schuster. Are there enough injective sets? *Studia Logica*, 101(3):467–482, 2012.

[2] Tom de Jong and Martín Hötzel Escardó. On small types in univalent foundations. *Logical Methods in Computer Science*, Volume 19(2), 2023.

[3] Martín H. Escardó. Infinite sets that satisfy the principle of omniscience in any variety of constructive mathematics. *The Journal of Symbolic Logic*, 78(3):764–784, 2013.

[4] Martín H. Escardó and Cory M. Knapp. Partial elements and recursion via dominances in univalent type theory. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, volume 82 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

[5] Martín Hötzel Escardó. Compact, totally separated and well-ordered types in univalent mathematics. Abstract for a talk at the *25th International Conference on Types for Proofs and Programs (TYPES)*. Available at https://martinescardo.github.io/papers/compact-ordinals-Types-2019-abstract.pdf, 2019.

[6] Martín Hötzel Escardó. Injective types in univalent mathematics. *Mathematical Structures in Computer Science*, 31(1):89–111, 2021.

[7] Martín Hötzel Escardó and Tom de Jong. Injective types in univalent mathematics in Agda notation. Agda development, HTML rendering available at: https://www.cs.bham.ac.uk/~mhe/TypeTopology/InjectiveTypes.index.html, code available at: https://github.com/martinescardo/TypeTopology, Since 2014.

[8] M. P. Fourman and A. Ščedrov. The "world's simplest axiom of choice" fails. *Manuscripta Mathematica*, 38(3):325–332, 1982.

[9] Håkon Robbestad Gylterud. From multisets to sets in homotopy type theory. *The Journal of Symbolic Logic*, 83(3):1132–1146, 2018.

[10] Håkon Robbestad Gylterud. Multisets in type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):1–18, 2019.

[11] Peter T Johnstone. *Sketches of an Elephant A Topos Theory Compendium*, volume 2 of *Oxford Logic Guides*. Oxford Science Publications, 2002.

[12] Zoltan A. Kocsis. Apartness relations between propositions. *Mathematical Logic Quarterly*, 70(4):414–428, 2024.

[13] Nicolai Kraus, Martín Hötzel Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of anonymous existence in Martin-Löf Type Theory. *Logical Methods in Computer Science*, 13(1), 2017.

[14] Michael Rathjen. *Proof Theory of Constructive Systems: Inductive Types and Univalence*, volume 13 of *Outstanding Contributions to Logic*, pages 385–419. Springer, 2017.

[15] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[16] Egbert Rijke, Michael Shulman, and Bas Spitters. Modalities in homotopy type theory. *Logical Methods in Computer Science*, 16(1), 2020.

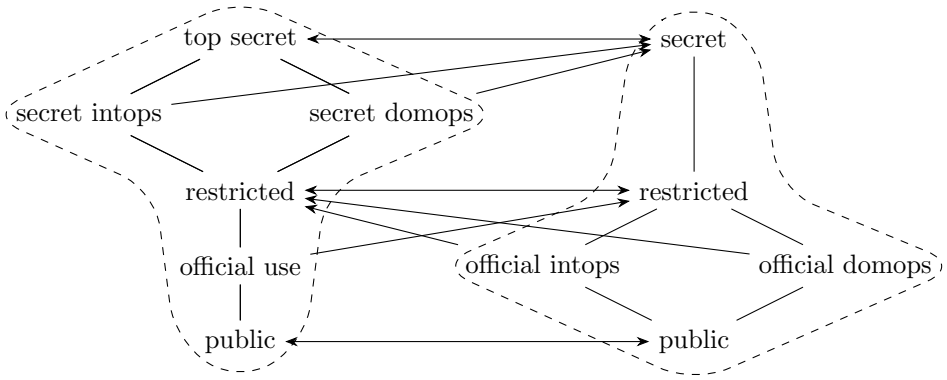
[17] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

A Coq Formalization of Lagois Connections for Secure Information Flow

Casper Ståhl, Levs Gondelmans,
René Rydhof Hansen, and Danny Bøgsted Poulsen

Aalborg University, Aalborg, Denmark
cstahl20@student.aau.dk, {lego,rrh,dannybpoulsen}@cs.aau.dk

Consider the situation illustrated below, in which two organisations wish to communicate in a secure way without compromising their respective local information flow policies, as defined using *security lattices* [4]. This becomes a matter of figuring out how to map security levels in one organisation to security levels in the other organisation and vice versa, while taking the intended communication pattern into account to ensure that no information can be leaked in one organisation by sending it on a "roundtrip" through the other organisation:



While the local security policies can be enforced, e.g., by type systems [6] or static analysis [3], it is non-trivial to prove that a given mapping between security levels is secure, i.e., that the local information security policies are not compromised even when taking external communication into account. Bhardwaj and Prasad propose the use of *Lagois connections* (as defined by Melton [5]) as a framework defining such mappings in a way that is secure by design, yet without mapping security levels to unnecessarily high counterparts [1, 2] and thereby avoid "label creep". Lagois connections are very similar to the more commonly known Galois connections with a slight change:

Definition 1 (Lagois connection)**.** A poset system (P, f, g, Q) is called an *(increasing) Lagois connection* iff $p \leq gf(p)$ for all $p : P$ (**LC1**), $q \leq fg(q)$ for all $q : Q$ (**LC2**), $fgf(p) = f(p)$ for all $p : P$ (**LC3**), and $gfg(q) = g(q)$ for all $q : Q$ (**LC4**).

With this definition in mind, flow is permitted between the two systems from p: P to q: Q when $f(p) \leq q$ and analogously the other way around. As an example, in the figure above information is allowed to flow from "secret intops" to "top secret" via the lattice itself but also by a round trip of the Lagois connection via "secret". The fact that f and g are monotone, by virtue of (P, f, g, Q) being a poset system, ensures *one way security* in the sense that policies are preserved post mapping, or more precisely $p \leq p'$ implies $f(p) \leq f(p')$ for all $pp' : P$ (analogous for g). **LC1** and **LC2** ensures back and forth security, in the sense that mapping

a label $p : P$ back and forth resulting in $gf(p)$, respects the local policy in the sense that $p \leq gf(p)$ (analogous for fg). In relation to information flow, **LC3** and **LC4** mainly ensure that the mappings are *precise* and that continued back and forth communication immediately converges after one round trip.

Our contribution is a Coq formalization of Lagois connections and related theory for use in secure information flow[1]. In particular, we have formalised the results from the seminal work of Melton et al. [5] that are of interest for secure information flow [1]. Further, we have used this formalisation to develop a formal variation of the type system developed by Bhardwaj and Prasad [1, 2] and have proved, formally in Coq, that that our variation is sound with respect to non-interference. Since our primary interest is secure information flow, the formalisation is limited to *security lattices*, that is, finite inhabited lattices, allowing for a more direct representation of many results by Melton et al. More importantly this moves us in a direction where Lagois connections can automatically be established and proved within Coq.

Our main result is a formal proof of soundness for a type system very similar to the one presented by Bhardwaj and Prasad [1, 2], i.e., one that is better suited for formal proofs and conjectured to be equivalent:

Theorem 1. *For a Lagois connection (P, f, g, Q), and an adversarial residing at level $p : P$ and $q : Q$ such that $p = g(q)$ and $q = f(q)$. If for a program s, environments ν, ν_f, ν', ν'_f and μ, μ_f, μ', μ'_f belonging to the organizations of P and Q respectively:*

1. *starting in environment (ν, μ), executing s evaluates to (ν_f, μ_f),*
2. *starting in environment (ν', μ'), executing s evaluates to (ν'_f, μ'_f),*
3. *s can be given security type (p', q'), and*
4. *$\nu(v) = \nu'(v)$ for all v with a security type $p'' : P$ such $p'' \leq p$ and $\mu(v) = \mu'(v)$ for all w with a security type $q'' : Q$ such $q'' \leq q$.*

Then $\nu_f(v) = \nu'_f(v)$ for all v with a security type $p'' : P$ such $p'' \leq p$ and $\mu_f(v) = \mu'_f(v)$ for all w with security type $q'' : Q$ such $q'' \leq q$.

Intuitively, this result states that for two systems communicating over channels that respect an underlying Lagois connection, *non-interference* will be guaranteed in each system, even taking the communication into account. I.e., no information is leaked, even if has been on a "roundtrip" through the other system. Essential to proving this result, the type system additionally checks if the Lagois connection is respected when communication takes place.

In addition to providing a formalisation of existing work, our approach allowed us to write certain proofs in a more direct style, simplifying working with both proofs and formalisation. Furthermore, we argue that our formalisation is a solid foundation for further work on both Lagois connections and the use of these for secure information flow. This is exemplified in our current work, using (and extending) the framework to investigate and formalise secure information flow between more than two systems as current methods [2] for extending the framework do not do so nicely: Composing Lagois connections $(P, f, g, Q), (Q, \hat{f}, \hat{g}, R)$ in a chain as $\mathbf{L} = (P, \hat{f}f, \hat{g}g, R) = (P, f, g, Q) \circ (Q, \hat{f}, \hat{g}, R)$ is secure in the sense that $\mathbf{L}$ satisfies both **LC1** and **LC2**, but such compositions may not be precise and may not converge (quickly). For example, $\mathbf{L}$ is only a Lagois connection iff $\hat{g}\hat{f}f[P] \subseteq f[P]$ and $fg\hat{g}[R] \subseteq \hat{g}[R]$ (Theorem 3.22 in [5]). Further, we have observed chains of Lagois connections $\mathbf{L} = (P, f, g, Q) \circ (Q, \hat{f}, \hat{g}, R)$

[1] The formalisation is available at https://github.com/CasperStaahl/TYPES-2025-formalization-preview

that are lossy, in the sense that the established Lagois connection $\mathbf{L}$ is less precise than one that could be established between P and R directly even though (P, f, g, Q) and $(Q, \hat{f}, \hat{g}, R)$ are as precise as possible.

In our ongoing work we explore ways to mitigate this limitation by considering Lagois connections over communication topologies given by undirected graphs. Not all topologies "automatically" give rise to secure communications, but some do and we are currently working to characterise such topologies.

References

[1] Chandrika Bhardwaj and Sanjiva Prasad. "Only connect, securely". In: *Formal Techniques for Distributed Objects, Components, and Systems: 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 39*. Springer. 2019, pp. 75–92.

[2] Chandrika Bhardwaj and Sanjiva Prasad. "Secure information flow connections". In: *Journal of Logical and Algebraic Methods in Programming* 127 (2022), p. 100761.

[3] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252.

[4] Dorothy E Denning. "A lattice model of secure information flow". In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[5] Austin Melton, Bernd SW Schröder, and George E Strecker. "Lagois connections—a counterpart to Galois connections". In: *Theoretical Computer Science* 136.1 (1994), pp. 79–107.

[6] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. "A sound type system for secure flow analysis". In: *Journal of computer security* 4.2-3 (1996), pp. 167–187.

Coq proof of the fifth Busy Beaver value

Tristan Stérin*[1,2] and The bbchallenge Collaboration[1]

[1] bbchallenge.org, see credits, bbchallenge@bbchallenge.org
[2] prgm.dev, Paris, France tristan@prgm.dev

Abstract

We prove that $S(5) = 47,176,870$. The Busy Beaver value $S(n)$ is the maximum number of steps a halting n-state 2-symbol Turing machine can perform from the all-0 tape before halting and S was historically introduced as one of the simplest examples of a noncomputable function. Using the Coq proof assistant, we enumerate 181,385,789 5-state Turing machines, and for each, decide whether it halts or not. Our result marks the first determination of a new Busy Beaver value in over 40 years, leveraging Coq's computing capabilities and demonstrating the effectiveness of collaborative online research.

Introduced by Tibor Radó in 1962 as part of the *Busy Beaver game* [15], $S(n)$ is the maximum number of steps that a halting n-state 2-symbol Turing machine[1] can do from the all-0 tape before halting. The function S is noncomputable as it would otherwise allow to solve the halting problem: run an n-state machine for $S(n) + 1$ steps and, if it has not halted by then, you know it will never halt.

As S in noncomputable, there is no algorithm that computes *all* of its values, but one can still try to determine *some* of them. We can get a sense of how hard this is by noting that, for instance, there is a 25-state 2-symbol Turing machine [4, 7] that iterates all even natural numbers in search of a counterexample to Goldbach's conjecture[2], and halts iff one is found (i.e. iff the conjecture is false), which means that proving the value of $S(\geq 25)$ is at least as hard as proving or disproving Goldbach's conjecture. Worse, this idea can be generalised to any other Π_1 statement[3], such as Riemann Hypothesis or ZF's consistency which have been respectively reduced to $S(744)$ and $S(748)$ [20, 19, 14, 19, 1, 6].

Nonetheless, researchers have embarked on the quest of finding what is the biggest value of S we can know. Motivations include: (i) trying to find the simplest open problem in mathematics on the Busy Beaver scale by exhibiting an n-state Turing machine for which it is unknown whether it halts or not but $S(n-1)$ is known (i.e. the halting status of all $n-1$-state Turing machines is known) and (ii) studying compute *in the wild* by discovering non-human-engineered algorithms that do *quirky things* in a *quirky way*.

Prior to this work, only the four first values of S had been proved: $S(1) = 1$, $S(2) = 6$ [15], $S(3) = 21$ [9], $S(4) = 107$ [3]. After some early attempts in the 1960s and 1970s [12, 13], the search for $S(5)$ took a turn in 1989 when Marxen and Buntrock found a new 5-state champion[4] (i.e. a 5-state machine with bigger step-count than previously known ones) achieving 47,176,870 steps [10], thus showing $S(5) \geq 47{,}176{,}870$. In 2020, after thirty years without improvements, Aaronson conjectured that there was no better 5-state machine, i.e. $S(5) = 47{,}176{,}870$ [1].

Our main result is to prove this conjecture, using the Coq proof assistant[5] [17]:

Theorem (`Lemma BB5_value`)**.** $S(5) = 47{,}176{,}870$.

The proof, called Coq-BB5, is available at https://github.com/ccz181078/Coq-BB5.

*Presenting author

[1] Turing machines with one bi-infinite tape and allowing undefined transitions.

[2] The conjecture states that every even natural number $n > 2$ can be written as the sum of two primes. This is one of the oldest open problems in mathematics.

[3] i.e. a logical statement of the form "For all x, $\phi(x)$" with ϕ using only bounded quantifiers meaning that a computer can check $\phi(x)$ in finite time for any x.

[4] https://bbchallenge.org/1RB1LC_1RC1RB_1RD0LE_1LA1LD_1RZ0LA

[5] The Coq proof assistant has been renamed Rocq.

$S(5)$ pipeline	Nonhalt	Halt	Total decided
1. Loops	126,994,099	48,379,711	175,373,810
2. n-gram Closed Position Set (NGramCPS)	6,005,142	0	6,005,142
3. Repeated Word List (RepWL)	6,577	0	6,577
4. Finite Automata Reduction (FAR)	23	0	23
5. Weighted Finite Automata Reduction (WFAR)	17	0	17
6. Long halters (simulation up to 47,176,870 steps)	0	183	183
7. Sporadic Machines, individual proofs	13	0	13
8. Reduction to `1RB`	14	0	14
Total	133,005,895	48,379,894	181,385,789

Table 1: Approximation of the $S(5)$ pipeline as implemented in Coq-BB5. All the 181,385,789 enumerated 5-state machines are decided by this pipeline, which solves $S(5) = 47{,}176{,}870$.

In this talk, we will present Coq-BB5 and describe, at various technical levels, how the proof works. The overall structure of the proof is as follows: the proof enumerates 5-state machines in *Tree Normal Form* (**TNF**) [2, 3, 10]. TNF essentially consists in enumerating partially-defined Turing machines starting from a machine with no transitions defined, each enumerated machine is passed through a pipeline of proof techniques mainly consisting of *deciders* which are algorithms trying to decide whether the machine halts or not [6]:

1. If the machine halts, i.e. meets an undefined transition, a new subtree of machines is visited for all the possible ways to fill the undefined transition.
2. If the machine does not halt, it is a leaf of the TNF tree.

The TNF enumeration terminates when all leafs have been reached, i.e. all the enumerated Turing machines have been decided and there are no more halting machines to expand into subtrees. The proof enumerates 181,385,789 machines[7] in about 45 minutes[8], and runs the pipeline summarised in Table 1 on each of them. All the algorithms (TNF enumeration and deciders) are programmed and proved correct in Coq.

The deciders essentially leave 13 *Sporadic Machines* undecided, for which we provide individual Coq proof of correctness [11, 18]. Among them, the monstruous *Skelet #1*[9], which is a translated loop (i.e. eventually repeats the same pattern translated in space) that only starts looping after 10^{51} steps with a period of more than 8 billion steps [8]. This machine is named after Georgi Georgiev (a.k.a Skelet) who first found it, as well as 42 other arduous ones, back in 2003 [5]. After all the enumerated machines are proved halting/nonhalting, the proof concludes $S(5) = 47{,}176{,}870$: Marxen and Buntrock's champion is the winner of the fifth Busy Beaver competition.

The talk will also discuss meta aspects of this result such as how were proof assistants used in practice and how the research was conducted within The bbchallenge Collaboration, a self-organised, international, online community of scientists (most without academic affiliation) gathered around the `bbchallenge.org` platform [16].

[6] We don't use the word "decider" in the usual sense of theoretical computer science, formally we mean an algorithm that (i) takes as input a Turing machine, (ii) is guaranteed to finish, and (iii) returns either `HALT`, `NONHALT` or `UNKNOWN`. Proving that a decider is correct means that, if the decider returns `HALT`/`NONHALT` then, indeed, the machine halts/does not halt.

[7] The list of enumerated machines, with halting status and decider information is made available at: `https://docs.bbchallenge.org/CoqBB5_release_v1.0.0/`.

[8] When compiled in parallel using `native_compute`. Unparallelised compilation with `vm_compute` takes about 13 hours (on a Macbook Pro M3 Max).

[9] `https://bbchallenge.org/1RB1RD_1LC0RC_1RA1LD_0RE0LB_---1RC`

The bbchallenge Collaboration (credits). The following contributions resulted in the determination of the fifth Busy Beaver value: mxdys (Coq-BB5, Loops, RepWL); Nathan Fenner, Georgi Georgiev a.k.a Skelet, savask, mxdys (NGramCPS); Justin Blanchard, Mateusz Naściszewski, Konrad Deka (FAR); Iijil (WFAR); mei (`busycoq`); Shawn Ligocki, Jason Yuen, mei (Sporadic Machines "Shift Overflow Counters"); Shawn Ligocki, Pavel Kropitz, mei (Sporadic Machine "Skelet #1"); savask, Chris Xu, mxdys (Sporadic Machine "Skelet #17"); Shawn Ligocki, Dan Briggs, mei (Sporadic Machine "Skelet #10"); Justin Blanchard, mei (Sporadic Machines "Finned Machines"); Shawn Ligocki, Daniel Yuan, mxdys, Rachel Hunter ("Cryptids"); Yannick Forster, Théo Zimmermann (Coq review); Yannick Forster (Coq optimisation); Tristan Stérin (bbchallenge.org); Tristan Stérin, Justin Blanchard (paper writing).

References

[1] S. Aaronson. The Busy Beaver Frontier. *SIGACT News*, 51(3):32–54, Sept. 2020. https://www.scottaaronson.com/papers/bb.pdf.

[2] A. H. Brady. *Solutions of restricted cases of the halting problem applied to the determination of particular values of a non-computable function*. PhD thesis, Oregon State University, 1964.

[3] A. H. Brady. The determination of the value of rado's noncomputable function $\Sigma(k)$ for four-state turing machines. *Mathematics of Computation*, 40(162):647–665, 1983.

[4] Code Golf Addict. list27.txt. https://gist.github.com/anonymous/a64213f391339236c2fe31f8749a0df6, 2016.

[5] G. Georgiev. Busy Beaver prover - `bbfind`. https://skelet.ludost.net/bb/, 2003. Accessed: 2024-11-25.

[6] Johannes Riebel. The Undecidability of BB(748). Bachelor's thesis, 2023. https://www.ingo-blechschmidt.eu/assets/bachelor-thesis-undecidability-bb748.pdf.

[7] Y. Leng. lengyijun/goldbach_tm: 25-state turing machine, Jan. 2025.

[8] S. Ligocki. Skelet #1 is infinite ... we think. Blog: https://www.sligocki.com/2023/03/13/skelet-1-infinite.html. Accessed: 2024-02-11.

[9] S. Lin. *Computer studies of Turing machine problems*. PhD thesis, Ohio State University, Graduate School, 1963.

[10] H. Marxen and J. Buntrock. Attacking the Busy Beaver 5. *Bull. EATCS*, 40:247–251, 1990.

[11] mei. busycoq. https://github.com/meithecatte/busycoq/blob/master/verify/Skelet1.v, 2023.

[12] P. Michel. The Busy Beaver Competitions. https://bbchallenge.org/~pascal.michel/bbc.html [Accessed 04-08-2024].

[13] P. Michel. The Busy Beaver Competition: a historical survey. Technical report, Sept. 2019.

[14] S. O'Rear. metamath - ZF. https://github.com/sorear/metamath-turing-machines/blob/master/zf2.nql, 2017.

[15] T. Radó. On non-computable functions. *Bell System Technical Journal*, 41(3):877–884, 1962. https://archive.org/details/bstj41-3-877/mode/2up.

[16] T. Stérin. bbchallenge.org, initial release, Mar. 2022.

[17] T. C. D. Team. The coq proof assistant, Dec. 2024.

[18] C. Xu. Skelet #17 and the fifth Busy Beaver number, 2024.

[19] A. Yedidia and S. Aaronson. A relatively small Turing machine whose behavior is independent of set theory. *Complex Systems*, 25(4):297–328, Dec. 2016.

[20] S. A. Yuri Matiyasevich, Stefan O'Rear. metamath - Riemann Hypothesis. https://github.com/sorear/metamath-turing-machines/blob/master/riemann-matiyasevich-aaronson.nql, 2016.

Mechanizing logical relations

Josselin Poiret

Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, F-44000 Nantes, France

Meta-theoretical studies of type theories often rely on realizability techniques, including the so-called "logical relations", to establish interesting properties of the systems. The prototypical examples of such results are normalization for simply typed lambda calculus with Tait's computability technique [Tai67], then System F with Girard's reducibility candidates [Gir72], the same year as Martin-Löf Type Theory (MLTT) with [Mar98].

However, type theorists and practitioners did not stop with MLTT, and newer, more convenient systems were created and gradually refined for specific applications. Today, a plethora of systems are used to prove and program, with examples as diverse as Homotopy Type Theory (HoTT) and Cubical Type Theory for synthetic homotopy theory, Multi-Modal Type Theory for the internal logic of specific toposes, or 2-Level Type Theory for interaction between HoTT and a type theory with uniqueness of identity proofs (UIP). With a proliferation of new, more exotic type systems, the demand for meta-theoretical verification has grown significantly.

Proving properties of such systems has also gotten harder, proportionally with their expressive power and with the expectation that such proofs must now be formalized. Some formalization efforts have focused on the engineering part of such proofs, while also providing a solid base for experimentation with new features. Projects such as MetaCoq [Soz+20], Agda Core or Lean4Lean [Car24] focus on the idiosyncrasies of specific proof assistants, while LogRel-MLTT [AÖV18] and its direct descendant LogRel-Coq [Adj+24] study a more idealized version of MLTT.

In the case of the latter, we have been trying to simplify its implementation, as some design choices for the propagated invariants can sometimes feel very arbitrary and redundant. We are also trying to precisely understand what comparisons can be made with high level abstractions like proofs by normalisation by evaluation [Fio22] and Synthetic Tait Computability (STC) [Ste21] and whether some of their tools can be used in our developments. I will present some work-in-progress improvements, proof techniques, as well as future directions for the development, highlighting our expectations for the project.

What's in a logical relation formalization? Both of LogRel-MLTT and LogRel-Coq follow a similar pattern of defining realizers in an inductive-recursive manner (with a trick when encoding in Rocq). What kind of information these realizers contain is not fixed though! The definition of realizers is parametrized by another notion of "semantic typing", which has to follow quite a lot of axioms for the fundamental lemma to go through. The choice of semantic typing influences what kind of result one gets via the logical relation argument, but also constrains what kind of type theory we can interpret.

In the current development, to prove the decidability of typing and conversion, this semantic typing is instantiated by the graph of a bidirectional typing algorithm. However, that system does not satisfy the axioms mentioned above at first glance. To prove that it does, one needs to go through the whole fundamental lemma with a different choice of semantic typing, which seems very inefficient (and not very principled). We would like to only instantiate the construction once with a system that gives us the expected corollaries directly.

Canonical derivations Standard presentations of dependent type theories include the general application inference rule and its congruence counterpart

$$\frac{\Gamma \vdash f : \Pi\, A\, B \qquad \Gamma \vdash a : A}{\Gamma \vdash f : B[\uparrow a]}\ \textsc{App} \qquad \frac{\Gamma \vdash f \equiv g : \Pi\, A\, B \qquad \Gamma \vdash a \equiv b : A}{\Gamma \vdash f\ a \equiv g\ b : B[\uparrow a]}\ \textsc{AppCong}$$

These inference rules, more specifically AppCong, cause derivations of conversion to not be unique, as for example $\Gamma \vdash (\lambda\ x : A, x)\ y \equiv (\lambda\ x : A, x)\ y : A$ can be derived because of either AppCong directly, or by a transitive combination of β-reductions. For the same reason, derivations do not contain normal form information, as applications can be typed without reducing them.

A different presentation of dependent type theory can also be given, with a work-in-progress formalization, whose derivations all faithfully reflect β-normal η-long reduction information. This can be achieved by avoiding the general App and AppCong rules, and instead always relying on reduction, even for typing. This technique is reminiscent of cut-free presentations of first-order logic, and is behind some bidirectional type systems, including the one present in LogRel-Coq, although only for the conversion judgements [Len21].

Once the system is established, a key property to show first is that derivations are indeed unique, as well as inferred types. With this uniqueness result, basic properties like transitivity of conversion are shown with straightforward strong inductions. Then, the crux of the realizability technique should be able to show that general elimination rules are admissible for our new system.

Proof-engineering techniques in Rocq As is usual in meta-theory, proofs are conceptually clear but require a lot of tedious work. I will also mention a couple of techniques that we employ to stay within Rocq's restrictions while keeping our proofs manageable.

One drastic change is to only use partial equivalence relations (PERs) for the judgements of our type theory, and create a completely binary version of the system. As an example, the usual judgement $\Gamma \vdash a : A$ would correspond to $\Gamma \equiv \Gamma \vdash a \equiv a : A \equiv A$, and $\Gamma \vdash a \equiv b : A$ to $\Gamma \equiv \Gamma \vdash a \equiv b : A \equiv A$. Though this might seem like a needless increase in complexity, this lets us treat all rules uniformly while not repeating ourselves when proving properties about both the typing rules and the congruence rules.

More pragmatically, mutual inductive types in Rocq are quite impractical to use, especially since one needs to manually generate the combined schemes, as well as provide numerous induction motives on use. Instead, I use a single indexed inductive type, leading to a net gain in conciseness when writing down the type signatures of properties. Using the unary parametricity translation of that inductive lets me define a strong induction principle, along the lines of [Tas19]. With these generalizations, proving properties of our canonical system is relatively speedy.

References

[Adj+24] 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, London, UK, January 15-16, 2024*. Ed. by Amin Timany et al. ACM, 2024, pp. 230–245. DOI: 10.1145/3636501.3636951. URL: https://doi.org/10.1145/3636501.3636951.

[AÖV18] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. "Decidability of conversion for type theory in type theory". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 23:1–23:29. DOI: 10.1145/3158111. URL: https://doi.org/10.1145/3158111.

[Car24] Mario Carneiro. "Lean4Lean: Towards a formalized metatheory for the Lean theorem prover". In: *CoRR* abs/2403.14064 (2024). DOI: 10.48550/ARXIV.2403.14064. arXiv: 2403.14064. URL: https://doi.org/10.48550/arXiv.2403.14064.

[Fio22] Marcelo Fiore. "Semantic analysis of normalisation by evaluation for typed lambda calculus". In: *Math. Struct. Comput. Sci.* 32.8 (2022), pp. 1028–1065. DOI: 10.1017/S0960129522000263. URL: https://doi.org/10.1017/S0960129522000263.

[Gir72] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur". PhD thesis. Université Paris VII, June 1972.

[Len21] Meven Lennon-Bertrand. "Complete Bidirectional Typing for the Calculus of Inductive Constructions". In: *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 24:1–24:19. DOI: 10.4230/LIPICS.ITP.2021.24. URL: https://doi.org/10.4230/LIPIcs.ITP.2021.24.

[Mar98] Per Martin-Löf. "An Intuitionistic Theory of Types". In: *Twenty Five Years of Constructive Type Theory*. Ed. by Giovanni Sambin and Jan M. Smith. Also known as MLTT72, published version of an unpublished 1972 note. Clarendon Press, 1998, pp. 127–172.

[Soz+20] Matthieu Sozeau et al. "The MetaCoq Project". In: *J. Autom. Reason.* 64.5 (2020), pp. 947–999. DOI: 10.1007/S10817-019-09540-0. URL: https://doi.org/10.1007/s10817-019-09540-0.

[Ste21] Jonathan Sterling. "First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory". Version 1.1, revised May 2022. PhD thesis. Carnegie Mellon University, 2021. DOI: 10.5281/zenodo.6990769.

[Tai67] William W. Tait. "Intensional Interpretations of Functionals of Finite Type I". In: *J. Symb. Log.* 32.2 (1967), pp. 198–212. DOI: 10.2307/2271658. URL: https://doi.org/10.2307/2271658.

[Tas19] Enrico Tassi. "Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq". In: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 29:1–29:18. DOI: 10.4230/LIPICS.ITP.2019.29. URL: https://doi.org/10.4230/LIPIcs.ITP.2019.29.

Geometric Reasoning in Lean: from Algebraic Structures to Presheaves

Kenji Maillard[1] and Yiming Xu[2]

[1] Inria, Nantes, France
[2] Ludwig-Maximilian Universität, München, Germany

Abstract

Algebraic theories such as semigroups, monoids, rings or Heyting algebras can easily be described in dependent type theories, such as that of Lean, by packing together sorts, operations and equations. Abstractly, these theories should be interpretable in many different settings beyond bare types. Leveraging semantics of geometric logic in presheaf categories, i.e. categories of "varying sets", we explore the potential of interpreting such algebraic theories in these extended settings.

Motivation & Goals By packing together types, operations and (equational) proofs, dependent type theories provide direct representation of algebraic structures. For instance, the notion of semigroup is defined in Lean's mathlib library [2] as (a variant of)

```
class Semigroup (G : Type u) where
  * : G -> G -> G
  /-- Multiplication is associative -/
  protected mul_assoc : forall a b c : G, a * b * c = a * (b * c)
```

Algebraic structures also admit an alternative approach within dependent type theories by presenting them through equational theories. The benefits of this latter indirect point of view is that we can consider models of such theories in settings beyond that of bare types as carriers of sorts. In our work, we leverage geometric theories to represent such algebraic structures and use their well-studied categories of models in presheaf categories, e.g. categories of families of sets parametrized by some base category. Following [1], we formalize a notion of geometric theories $\mathcal{T}$ in Lean [3] and construct the category $\mathcal{T}$-Mod($\mathcal{C}$) of $\mathcal{T}$-models, using the notation of [1, Definition 1.2.12], in any presheaf category Func($\mathcal{C}^{\mathrm{op}}, \mathcal{S}et$), functorially with respect to $\mathcal{C}$. We then explore the correspondence between the direct and indirect presentations of algebraic structures, with the hope of being able to transfer direct proofs about the former to constructions applicable on models in any presheaf category. Our Lean formalization is available at https://github.com/kyoDralliam/model-theory-topos.

Geometric Logic, Geometric Theories With an eye towards interpretations in presheaf categories, we choose to work with geometric logic, a slight extension of the $\wedge, \vee, =, \exists$ fragment of first-order logic with arbitrarily large disjunctions $\bigvee_{i \in I} \phi_i$. This logic is both expressive enough to represent many theories of interest, in particular any equational, algebraic theory, and admits interpretations in large classes of categories, notably in any (pre)sheaf topos. We define a deep embedding of single-sorted geometric logic in Lean parametrized by a signature collecting operations and predicates of finitary arity. The formalization employs well-scoped syntax and parametrizes geometric formulas with a small universe $(U, (I_k)_{k \in U})$ to deal with arbitrary disjunction $\bigvee_{i \in I} \phi_i$. A geometric theory $\mathcal{T}$ consists of a signature $\Sigma_{\mathcal{T}}$ together with a collection of axioms $(\phi_i \vdash_{\vec{x}_i} \psi_i)_i$ provided by sequents of geometric formulas over $\Sigma_{\mathcal{T}}$.

Interpreting into Presheaf Categories We formalize the notion of model of a geometric theory $\mathcal{T}$ in presheaves over a base category $\mathcal{C}$. Concretely, it consists of a collection of models in $\mathcal{S}et$ indexed by the objects of $\mathcal{C}$ equipped with structure-preserving maps induced by the morphisms of $\mathcal{C}$. This formalization choice goes much beyond the specific case of $\mathcal{S}et$ but uses that most constructs are lifted pointwise from their counterpart $\mathcal{S}et$, a simplification that wouldn't be achievable in an arbitrary (Grothendieck) topos or a geometric category.

We establish the soundness of our interpretation of geometric logic, obtaining the following theorem by a straightforward induction:

```
theorem soundness {T : theory} (M:Mod T D) {n : RenCtx} (phi psi: fml T.sig n)
    (h:Hilbert.proof phi psi): InterpPsh.Str.model M.str (sequent.mk _ phi psi)
```

In words, given a model $\mathtt{M} \in \mathcal{T}\text{-Mod}(\mathcal{D})$, any proof $h : \phi \vdash_{x_1,\ldots,x_n} \psi$ induces a factorization $[\![\phi]\!]^{\mathtt{M}} \rightarrowtail [\![\psi]\!]^{\mathtt{M}}$ of the interpretation of formulas ϕ, ψ as subobjects of $\mathtt{M}^n$.

Rather than working with a single presheaf category each time in isolation, we establish the interaction between them: to transport interpretations of geometric formulas between presheaf categories, we formalize the construction of a functor $\mathcal{T}\text{-Mod}(\mathcal{D}) \to \mathcal{T}\text{-Mod}(\mathcal{C})$ from a functor $F : C \to D$ via pullback.

Connecting to Lean's Algebraic Structures We now turn to the interplay between Lean's algebraic structures (i.e. dependently typed records) and their encoding as geometric theories. We experiment with the simple examples of semigroups and monoids. For instance, the signature of semigroups consists of a single binary operation and no predicates, while the corresponding theory adds the associativity axiom as follows:

```
def semigroup_sig : monosig where
  ops := Unit
  arity_ops := fun \_ => 2
  preds := Empty
  arity_preds := Empty.rec

abbrev mul (t1 t2: tm semigroup_sig n)
    : tm semigroup_sig n :=
 .op () (fun i => [t1 , t2][i])
```

```
def assoc : sequent semigroup_sig where
  ctx := 3
  premise := .true
  concl := fml.eq (mul (.var 0) (mul (.var 1)
    (.var 2))) (mul (mul (.var 0) (.var 1))
    (.var 2))

def semigroup_thy : theory where
  sig := semigroup_sig
  axioms := [ assoc ]
```

As a sanity check, we prove that the category of models of semigroups over the trivial category with a single object is equivalent to Mathlib's category of semigroups.

Beyond relating algebraic structures with geometric theories, we also want to transport proofs. An interesting aspect of that process, is that even simple proofs about standard objects can turn into interesting properties once evaluated on non-trivial presheaf categories. We plan to investigate the specific case of monoids, building upon existing Lean proofs. We hope this example can inform us about the general recipe for extracting geometric proofs from proof scripts for other algebraic structures.

References

[1] Peter T. Johnstone. *Sketches of an Elephant: Volume 2*. Oxford University Press UK, Oxford, England, 2002.

[2] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Leonardo de Moura and Sebastian Ullrich. 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*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag.

Löb's Theorem and Provability Predicates in Rocq

Janis Bailitis[1,3], Dominik Kirst[2], and Yannick Forster[2]

[1] Saarland University, Saarbrücken, Germany
[2] Inria, Paris, France
[3] University of Oxford, Oxford, U.K.

Löb's theorem ([24], short: LT) states that in sufficiently strong formal systems such as Peano Arithmetic (PA), for a sentence φ we have $\mathsf{PA} \vdash \varphi$ if and only if $\mathsf{PA} \vdash \mathsf{prov}_{\mathsf{PA}}(\ulcorner \varphi \urcorner) \dot{\to} \varphi$, where the formula $\mathsf{prov}_{\mathsf{PA}}(x)$ is the standard provability predicate. It is a strengthening of Gödel's second incompleteness theorem (short: G2) which can be recovered via $\varphi := \bot$. Similar to the incompleteness theorems, the proof of LT is highly technical. Even for a fixed system such as PA, there are many different provability predicates of varying strengths. Not all of them qualify for LT, and formal reasoning about provability predicates is highly tedious.

For Gödel's first incompleteness theorem (short: G1), even in Rosser's [31] strengthening, these technical challenges can be avoided, as demonstrated by Kirst and Peters [20] who mechanise an abstract and computational proof of G1 [12] due to Kleene [21, 22] in Rocq [34]. They build their proof on the axiom Church's Thesis (CT) [23, 35], a well-understood axiom in constructive mathematics stating that quantifiers over functions in a constructive setting only range over computable functions. Fundamentally, their proof reduces G1 to the undecidability of provability in PA, mechanised by Kirst and Hermes [18].

For this abstract, we

1. mechanise the traditional proof of G1 via Carnap's diagonal lemma [6], using CT, with Rosser's [31] strengthening, complementing Kirst and Peters' computational mechanisation,
2. mechanise Tarski's theorem [33] about the undefinability of truth,
3. mechanise a proof of LT parameterised against a sufficiently strong provability predicate,
4. define a provability predicate and prove some of the necessary properties, all in Rocq,
5. extend Paulson's [28, 27] Isabelle mechanisation of a sufficiently strong predicate to yield LT.

In general, the abstract approach of Kirst and Peters does not extend to G2 or LT, because these theorems inherently rely on concrete implementation details of the underlying logical system, unlike G1. The results of this paper mechanised in the Rocq Prover [34] formally work in the Calculus of Inductive Constructions (CIC) [7, 26]. They rely on and are contributed to the Rocq Library for First-Order Logic [19] and the Rocq Library of Undecidability Proofs [11]. All proofs are constructive. This abstract is based on the first author's Bachelor's thesis [1] carried out in the group of Gert Smolka, advised by the other authors.

Mechanised synthetic computability. We use *synthetic computability theory* due to Richman, Bridges, and Bauer [29, 5, 2]. In synthetic computability theory, the usual notions from computability theory are defined without referring to a concrete model of computation. For instance, a predicate $P : X \to \mathbb{P}$ is said to be decidable iff there is a decider $f : X \to \mathbb{B}$ such that for all x, $P\,x$ holds iff $f\,x = \mathsf{true}$. If $\mathcal{O}(X) ::= \mathsf{Some}(x) \mid \mathsf{None}$ denotes the option type over X, we say that P is (recursively) enumerable if there is an enumerator $f : \mathbb{N} \to \mathcal{O}(X)$ such that for all x, $P\,x$ holds iff there exists n such that $f\,n = \mathsf{Some}(x)$. In type theory, synthetic computability has been developed by Forster [9] and colleagues. Of importance for this abstract is the standard fact [10] that $\lambda\varphi.\, T \vdash \varphi$ is (recursively) enumerable if T is.

Arithmetical Church's thesis. We define a variant of Church's thesis for Robinson arithmetic (Q) [30], a subsystem of PA. A formula φ is Σ_1 if it is of the form $\dot{\exists} x_1. \ldots. \dot{\exists} x_n. \psi$ and ψ does not use unbounded quantification. A theory T is called Σ_1-sound if $T \vdash \varphi$ implies $\mathbb{N} \vDash \varphi$ for all Σ_1-formulas φ. Church's thesis for Robinson arithmetic (CT_Q) states that for every function $f : \mathbb{N} \to \mathbb{N}$, there exists a binary Σ_1-formula $\varphi(x, y)$ such that for all n, it holds that $\mathsf{Q} \vdash \dot{\forall} y. \varphi[\overline{n}, y] \dot{\leftrightarrow} y \doteq \overline{f\, n}$. CT_Q is already employed by Kirst, Hermes, and Peters [14, 15, 20]. If applied to deciders and enumerators, CT_Q implies that certain predicates can be represented in Q as well [20, 14]. Most importantly, for any (recursively) enumerable $P : X \to \mathbb{P}$ and any Σ_1-sound $T \supseteq \mathsf{Q}$, there is a unary Σ_1-formula $\varphi(x)$ such that for all n, $P\, n$ holds iff $T \vdash \varphi[\overline{n}]$ (*weak representation*). Also, for any disjoint (recursively) enumerable $P, P' : X \to \mathbb{P}$ and consistent $T \supseteq \mathsf{Q}$, there is a unary Σ_1-formula $\varphi(x)$ such that for all n, both $P\, n$ implies $T \vdash \varphi[\overline{n}]$ and $P'\, n$ implies $T \vdash \dot{\neg}\varphi[\overline{n}]$ (*strong separation*).

Diagonal lemma and G1. A standard approach [4, 32] to prove G1 is to establish the diagonal lemma [6], stating that for all unary formulas $\varphi(x)$ there is a sentence G such that $\mathsf{Q} \vdash \varphi[\ulcorner G \urcorner] \dot{\leftrightarrow} G$, where $\ulcorner \cdot \urcorner$ is an encoding of formulas into closed terms. As a second step, the diagonal lemma is used to diagonalise against a provability predicate. We observe that the diagonal lemma readily follows from CT_Q. G1 then easily follows from (recursive) enumerability of $\lambda\varphi. T \vdash \varphi$ with a brief application of weak representability and diagonalisation. We also mechanise the following strengthening of G1 following Rosser [31].

Theorem 1 (G1 [12])**.** *Let $T \supseteq \mathsf{Q}$ be (recursively) enumerable and consistent. Then there is an independent sentence for T.*

Here, the idea is to diagonalise against the negation of the formula obtained from strong separability applied to $\lambda\varphi. T \vdash \varphi$ and $\lambda\varphi. T \vdash \dot{\neg}\varphi$. Similar reasoning also gives rise to Tarski's theorem [33].

Theorem 2 (Tarski [33])**.** *There is no first-order formula $\mathsf{true}_\mathbb{N}(x)$ such that $\mathbb{N} \vDash \varphi$ iff $\mathbb{N} \vDash \mathsf{true}_\mathbb{N}[\ulcorner \varphi \urcorner]$ for all sentences φ.*

For all these results, CT_Q is extremely helpful since instead of defining actual formulas for the provability predicates, only enumerability of provability is needed, which is easy to establish synthetically. The proofs thus reduce to the key insights gained from Gödel's and Tarski's work, making the proofs extremely concise. Without CT_Q, the results can still be shown if one assumes that T is μ-recursively enumerable, but then the proofs would become very tedious.

LT and internal vs external provability. Following a classification due to Feferman [8], we distinguish between *external* and *internal* provability predicates. An external provability predicate only has to correctly identify provable formulas, i.e. $T \vdash \varphi$ iff $T \vdash \mathsf{prov}_T[\ulcorner \varphi \urcorner]$ for all φ. CT_Q implies the existence of an external provability predicate, which is sufficient for Theorem 1. Mostowski [25], Bezboruah, and Shepherdson [3] observe that external predicates are too weak for LT by giving an external predicate for which G2 and hence also LT fails.

An internal provability predicate needs to additionally allow proving some deduction rules of the logical system as object level implications. This was made precise by Löb [24], based on previous work by Hilbert and Bernays [16]. The required properties are known as Hilbert-Bernays-Löb (HBL) derivability conditions, which are as follows, where φ, ψ are arbitrary formulas:

$$T \vdash \varphi \text{ implies } T \vdash \mathsf{prov}_T[\ulcorner \varphi \urcorner] \qquad \text{(necessitation)}$$

$$T \vdash \mathsf{prov}_T[\ulcorner \varphi \dot{\to} \psi \urcorner] \dot{\to} \mathsf{prov}_T[\ulcorner \varphi \urcorner] \dot{\to} \mathsf{prov}_T[\ulcorner \psi \urcorner] \qquad \text{(modus ponens rule)}$$

$$T \vdash \mathsf{prov}_T[\ulcorner \varphi \urcorner] \dot{\to} \mathsf{prov}_T[\mathsf{prov}_T[\ulcorner \varphi \urcorner]] \qquad \text{(internal necessitation)}$$

For an internal provability predicate, LT follows abstractly, and we mechanise this abstract proof in Rocq.

Theorem 3 (Löb [24])**.** *Let T be a theory admitting the diagonal lemma and let $\mathsf{prov}_T(x)$ satisfy the HBL conditions. Then $T \vdash \varphi$ iff $T \vdash \mathsf{prov}_T(\ulcorner \varphi \urcorner) \to \varphi$ for all sentences φ.*

Defining internal provability predicates. To do so, most authors arithmetise the notion of provability and define a complicated formula $\mathsf{prf}_T(x, y)$ such that $\mathsf{prf}_T(x, y)$ is provable iff y arithmetises a proof of the formula with code x. In this setting $\dot{\exists} y.\, \mathsf{prf}_T(x, y)$ characterises provability of x. Usually, a proof of a formula φ is encoded as a list of formulas representing the deduction of φ from the deduction rules. This is also how the standard provability predicate for PA is constructed.

While PA and related systems of arithmetic are strong enough to express the required list functions and to prove properties about these functions on the object level, choosing arithmetic to define provability predicates is not ideal. Instead, we prove the following:

Theorem 4. *There is an extension of* PA *with native function symbols for basic list functions to avoid some of the technicalities. In this system, there is an internal provability predicate satisfying necessitation and the modus ponens rule.*

This development is axiom-free. The verification of necessitation and the modus ponens rule heavily relies on a proof mode for first-order logic due to Koch [17], which made these mechanisations possible in the first place. As part of this proof, we contribute a mechanisation of Hilbert systems and a proof of its equivalence to natural deduction to the Rocq Library of First-Order Logic [19].

Paulson [28, 27] mechanises an internal provability predicate in HF set theory, easing arithmetisation, but the definition and the correctness proofs are still very arduous. We add the following to Paulson's Isabelle development:

Theorem 5. *Paulson's provability predicate is sufficient to deduce LT.*

Future work. Ultimately, we would like to obtain a Rocq mechanisation and verification of an internal provability predicate in a suitable system of first-order logic resembling PA. Besides doing the tedious work to prove internal necessitation for our predicate, it also seems promising to follow the approach by Halbach and Leigh [13] who give a system of first-order logic with function symbols for syntax manipulation which can express PA. These syntax functions seem helpful in the verification of internal necessitation.

In addition, it may be desirable to understand how strong a theory T of first-order arithmetic needs to be such that the HBL conditions for T's standard provability predicate can be proved. There may be research on this of which the authors are currently unaware.

References

[1] Janis Bailitis. Löb's Theorem and Provability Predicates in Coq, 2024. Bachelor's thesis. URL: https://www.ps.uni-saarland.de/~bailitis/bachelor.php.

[2] Andrej Bauer. First Steps in Synthetic Computability Theory. In Martín Hötzel Escardó, Achim Jung, and Michael W. Mislove, editors, *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2005, Birmingham, UK, May 18-21, 2005*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 5–31. Elsevier, 2005. doi:10.1016/J.ENTCS.2005.11.049.

[3] A. Bezboruah and John C. Shepherdson. Gödel's Second Incompleteness Theorem for Q. *The Journal of Symbolic Logic*, 41(2):503–512, 1976. doi:10.1017/S0022481200051586.

[4] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, 5th edition, 2007.

[5] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. London Mathematical Society Lecture Note Series. Cambridge University Press, 1987.

[6] Rudolf Carnap. *Logische Syntax der Sprache*. Schriften zur wissenschaftlichen Weltauffassung. Springer Berlin, Heidelberg, 1st edition, 1934.

[7] Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

[8] Solomon Feferman. Arithmetization of metamathematics in a general setting. *Fundamenta mathematicae*, 49:35–92, 1960.

[9] Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021. doi:10.22028/D291-35758.

[10] Yannick Forster, Dominik Kirst, and Gert Smolka. On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 38–51. ACM, 2019. doi:10.1145/3293880.3294091.

[11] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. *CoqPL 2020: The Sixth International Workshop on Coq for Programming Languages*, 2020.

[12] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik*, 38(1):173–198, 1931.

[13] Volker Halbach and Graham E. Leigh. *The Road to Paradox*. Cambridge University Press, 2024. doi:10.1017/9781108888400.

[14] Marc Hermes and Dominik Kirst. An Analysis of Tennenbaum's Theorem in Constructive Type Theory. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2022.9.

[15] Marc Hermes and Dominik Kirst. An Analysis of Tennenbaum's Theorem in Constructive Type Theory. *CoRR*, abs/2302.14699, 2023. arXiv:2302.14699, doi:10.48550/ARXIV.2302.14699.

[16] David Hilbert and Paul Bernays. *Grundlagen der Mathematik*, volume 2. Springer, Berlin, 1st edition, 1939.

[17] Johannes Hostert, Mark Koch, and Dominik Kirst. A Toolbox for Mechanised First-Order Logic. *The Coq Workshop*, 2021.

[18] Dominik Kirst and Marc Hermes. Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq. *Journal of Automated Reasoning*, 67(1):13, 2023. doi:10.1007/S10817-022-09647-X.

[19] Dominik Kirst, Johannes Hostert, Andrej Dudenhefner, Yannick Forster, Marc Hermes, Mark Koch, Dominique Larchey-Wendling, Niklas Mück, Benjamin Peters, Gert Smolka, and Dominik Wehr. A Coq Library for Mechanised First-Order Logic. *The Coq Workshop*, 2022.

[20] Dominik Kirst and Benjamin Peters. Gödel's Theorem Without Tears - Essential Incompleteness in Synthetic Computability. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2023.30.

[21] Stephen C. Kleene. Recursive Predicates and Quantifiers. *Transactions of the American Mathematical Society*, 53:41–73, 1943. doi:10.2307/2267986.

[22] Stephen C. Kleene. *Mathematical Logic*. Dover Publications, 1967.

[23] Georg Kreisel. Mathematical logic. *Lectures on Modern Mathematics*, 3:95–195, 1965.

[24] Martin H. Löb. Solution of a Problem of Leon Henkin. *The Journal of Symbolic Logic*, 20(2):115–118, 1955. doi:10.2307/2266895.

[25] Andrzej Mostowski. Thirty years of foundational studies: lectures on the development of mathematical logic and the study of the foundations of mathematics in 1930-1964. *Acta Philosophica Fennica*, 17:1–180, 1965.

[26] Christine Paulin-Mohring. Inductive Definitions in the system Coq - Rules and Properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993. doi:10.1007/BFB0037116.

[27] Lawrence C. Paulson. A Machine-Assisted Proof of Gödel's Incompleteness theorems for the Theory of Hereditarily Finite Sets. *The Review of Symbolic Logic*, 7(3):484–498, 2014. doi:10.1017/S1755020314000112.

[28] Lawrence C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015. doi:10.1007/S10817-015-9322-8.

[29] Fred Richman. Church's Thesis Without Tears. *The Journal of Symbolic Logic*, 48(3):797–803, 1983. doi:10.2307/2273473.

[30] Raphael Robinson. An essentially undecidable axiom system. *Proceedings of the International Congress of Mathematics*, pages 729–730, 1950.

[31] J. Barkley Rosser. Extensions of Some Theorems of Gödel and Church. *The Journal of Symbolic Logic*, 1(3):87–91, 1936. doi:10.2307/2269028.

[32] Peter Smith. *An Introduction to Gödel's Theorems*. Logic Matters, Cambridge, 2nd edition, 2020. URL: https://www.logicmatters.net/resources/pdfs/godelbook/GodelBookLM.pdf.

[33] Alfred Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica. Commentarii Societatis Philosophicae Polonorum*, 1:261 – 405, 1935. URL: https://www.sbc.org.pl/dlibra/publication/24411/edition/21615.

[34] The Coq Development Team. The Coq Proof Assistant, September 2023. doi:10.5281/zenodo.11551177.

[35] Dirk van Dalen and Anne S. Troelstra. *Constructivism in Mathematics*. Elsevier Science Publishers B.V., 1988.

Expansion in a Calculus with Explicit Substitutions

Ana Jorge Almeida, Sandra Alves, and Mário Florido

LIACC, Departamento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto

Term expansion was first defined in [8] to relate terms typed in an intersection type system with linear terms. Recently, new applications of *term expansion* include the relation with other substructural type systems (relevant and ordered type systems) [2] and quantitative types [3]. Here we define *term expansion* for a calculus with explicit substitutions and apply it to relate a λ-calculus with explicit substitutions [9] with a resource calculus [5], where the argument of a function is a bag of resources, that is, a multiset of terms.

1 Explicit substitutions

Although the λ-calculus [4, 6] is a convenient model for computational functions, it lacks the means for observing operational properties of the execution of such algorithms, mainly due to its implicit β-contraction, which is a meta-operation. There was a necessity to explicitly deal with substitutions, in order to bridge the gap between theory and implementation, allow efficient reduction in implementations, and avoid variable capture and scope issues [1].

Here we will be using a modification of the explicit substitution calculus presented in [9], the λxgc-calculus, which is an adaptation of $\lambda\sigma$ [1], that retains variable names instead of using indices *à la Bruijn* [7], and preserves strong-normalisation.

Definition 1 (λx-preterms)**.** The λx-preterms are the extension of the λ-preterms defined inductively by

$$M ::= x \mid \lambda x.M \mid MN \mid M < x := N >$$

In this calculus, explicit substitution is given highest precedence. It also has explicit garbage collection [10], which is useful and easy to specify using names.

Definition 2 (Evaluation in the λxgc-calculus)**.** The reduction is defined as $\xrightarrow[\text{bxgc}]{} = \xrightarrow[\text{b}]{} \cup \xrightarrow[\text{x}]{} \cup \xrightarrow[\text{gc}]{}$.

$$(\lambda x.M)N \xrightarrow[\text{b}]{} M < x := N > \qquad x < y := N > \xrightarrow[\text{xvgc}]{} x \text{ if } x \not\equiv y$$

$$x < x := N > \xrightarrow[\text{xv}]{} N \qquad M < x := N > \xrightarrow[\text{gc}]{} M \text{ if } x \notin fv(M)$$

$$(M_1 M_2) < x := N > \xrightarrow[\text{xap}]{} M_1 < x := N > M_2 < x := N >$$

$$\frac{M \xrightarrow[\text{bxgc}]{} M'}{MN \xrightarrow[\text{bxgc}]{} M'N} \qquad \frac{(\lambda x.M)N_2 \xrightarrow[\text{bxgc}]{} M' \quad \text{if } y \notin fv(N_2)}{((\lambda x.M) < y := N_1 >)N_2 \xrightarrow[\text{bxgc}]{} M' < y := N_1 >}$$

Example 1. Consider the λx-term $(\lambda x.xx)I$, where $I \equiv \lambda z.z$.

$$(\lambda x.xx)I \xrightarrow[\text{bxgc}]{} (xx) < x := I > \xrightarrow[\text{bxgc}]{} x < x := I > x < x := I > \xrightarrow[\text{bxgc}]{} II \xrightarrow[\text{bxgc}]{} z < z := I > \xrightarrow[\text{bxgc}]{} I$$

2 Term Expansion

We now relate expansion with a resource calculus [5], where the standard λ-calculus application MN, is denoted by MN^{∞}, to indicate that the argument N is always available for function M.

Our main focus now is to develop a relation between terms typable by idempotent intersection types and a subset of Boudol's language, which we will refer to as Λ^{∞}, allowing us to express only multiplicities ∞, hence each bag of resources is used without restriction.

Definition 3 (Boudol's terms)**.** The following terms are the syntax of Boudol's λ-calculus with multiplicities.

$$\begin{array}{rcll} M & ::= & x \mid \lambda x.M \mid (MP) \mid (M < P/x >) & \text{terms} \\ P & ::= & 1 \mid M \mid (P \mid P) \mid M^{\infty} & \text{bags of terms} \\ V & ::= & \lambda x.M \mid V < P/x > & \text{values} \end{array}$$

Definition 4 (Expansion)**.** Given a pair $M : \sigma$, where M is a λ-term and σ an intersection type, and a term N, we define a relation $\mathcal{E}(M : \sigma) \lhd N$, which we call *expansion*:

$$\begin{array}{rcll} \mathcal{E}(x : \tau) & \lhd & x & \\ \mathcal{E}(\lambda x.M : \tau_1 \cap \cdots \cap \tau_n \to \sigma) & \lhd & \lambda x.M^* & \text{if } x \in fv(M) \text{ and } \mathcal{E}(M : \sigma) \lhd M^* \\ \mathcal{E}(\lambda x.M : \tau \to \sigma) & \lhd & \lambda x.M^* & \text{if } x \notin fv(M) \text{ and } \mathcal{E}(M : \sigma) \lhd M^* \\ \mathcal{E}(MN : \sigma) & \lhd & (M^*(P_1^{m_1} \mid \cdots \mid P_k^{m_k})) & \text{if for some } k > 0 \text{ and } \tau_1 \ldots \tau_k \text{ such} \\ & & & \text{that } \mathcal{E}(M : \tau_1 \cap \cdots \cap \tau_k \to \sigma) \lhd M^* \\ & & & \text{and } \mathcal{E}(N : \tau_i) \lhd P_i^{m_i} \text{ for } 1 \leq i \leq k \\ \mathcal{E}(M < x := N >: \sigma) & \lhd & (M^* < (P_1^{m_1} \mid \cdots \mid P_k^{m_k})/x >) & \text{if for some } k > 0 \text{ and } \tau_1 \ldots \tau_k \text{ such} \\ & & & \text{that } \mathcal{E}(M : \tau_1 \cap \cdots \cap \tau_k \to \sigma) \lhd M^* \\ & & & \text{and } \mathcal{E}(N : \tau_i) \lhd P_i^{m_i} \text{ for } 1 \leq i \leq k \end{array}$$

Since we are working with Λ^{∞}, we have that for some $k > 0$ and $1 \leq i \leq k$, $m_i = \infty$.

Theorem 1 (Expansion and Multiplicities)**.** Given a λx-term M and a type σ, such that $\mathcal{E}(M : \sigma) \lhd M^*$, if $M \xrightarrow[\text{bxgc}]{}\!\!\!\!\!\rightarrow V_1$ then $M^* \twoheadrightarrow_{\text{B}} V_2$ and $\mathcal{E}(V_1 : \sigma) \lhd V_2$.

This theorem is proved by induction on the definition of expansion, and it shows that, if we evaluate a λx-term until it reaches a value, then we are able to expand that initial term, evaluate it in Boudol's system and its result is an expansion of the value obtained in the λx evaluation.

Example 2. Using the term in Example 1, we have

$$\mathcal{E}((\lambda x.xx)I : \sigma) \lhd ((\lambda x.(xx^{\infty}))I^{\infty})$$

because $\mathcal{E}(\lambda x.xx : ((\sigma \to \sigma) \cap \sigma) \to \sigma) \lhd \lambda x.(xx^{\infty})$ and $\mathcal{E}(I : (\sigma \to \sigma) \cap \sigma) \lhd I$.

$$\begin{array}{rcl} ((\lambda x.(xx^{\infty}))I^{\infty}) \to (xx^{\infty}) < I^{\infty}/x > \to (Ix^{\infty}) < I^{\infty}/x > & \to & z < x^{\infty}/z >< I^{\infty}/x > \\ & \to & x < x^{\infty}/z >< I^{\infty}/x > \\ & \to & I < x^{\infty}/z >< I^{\infty}/x > \equiv I \end{array}$$

and $\mathcal{E}(I : \sigma) \lhd I$.

3 Conclusion and Future Work

Here we have proved that there exists a relation between ACI-intersection types and a resource calculus that deals with multiplicities $m = \infty$ (infinitely available resources).

This serves as preliminary work towards proving that there exists a relation between AC-intersection types and a resource calculus of finite multiplicities. We also wish to look into an extension of this calculus that deals with α-conversion.

Acknowledgements This work was financially supported by: UID/00027 of the LIACC - Artificial Intelligence and Computer Science Laboratory - funded by Fundação para a Ciência e a Tecnologia, I.P./ MCTES through the national funds.

References

[1] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, 1989.

[2] Sandra Alves and Mário Florido. Structural rules and algebraic properties of intersection types. In *International Colloquium on Theoretical Aspects of Computing*, pages 60–77. Springer, 2022.

[3] Sandra Alves and Daniel Ventura. Quantitative weak linearisation. In *International Colloquium on Theoretical Aspects of Computing*, pages 78–95. Springer, 2022.

[4] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

[5] Gérard Boudol. The lambda-calculus with multiplicities. In *CONCUR'93: 4th Intrenational Conference on Concurrency Theory Hildesheim, Germany, August 23–26, 1993 Proceedings 4*, pages 1–6. Springer, 1993.

[6] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 33(2):346–366, 1932.

[7] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes mathematicae (proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[8] Mario Florido and Luis Damas. Linearization of the lambda-calculus and its relation with intersection type systems. *Journal of Functional Programming*, 14(5):519–546, 2004.

[9] Kristoffer H Rose. *Explicit substitution: tutorial & survey*. Computer Science Department, 1996.

[10] Kristoffer Høgsbro Rose. Explicit cyclic substitutions. In *International Workshop on Conditional Term Rewriting Systems*, pages 36–50. Springer, 1992.

Presheaves on Purpose

Conor McBride[1][2]

[1] University of Strathclyde
conor.mcbride@strath.ac.uk
[2] Quantinuum

Abstract

In current dependent type theories, we give types to the indices of inductive datatypes but we say very little about the structure of those index types. Categorically, they are treated as discrete, as the only structure automatically respected is equality. Here, I give a universe construction for datatypes indexed over small categories which are functorial by construction, hence presheaves, with a definition and proof given once for all.

1 Introduction

Whenever we find ourselves engineering coincidences, something is wrong. If we cannot articulate the design choices whose consequences we propagate, something is wrong. Let me show you something wrong. In Haskell, Agda, Idris, Lean or Coq (to name but a handful), I can write a function (pronounced 'thin')

$$\frac{t \,:\, \mathsf{Term}\ n \qquad \theta \,:\, n \sqsubseteq m}{t \uparrow \theta \,:\, \mathsf{Term}\ m} \qquad\qquad t \uparrow \iota = t \qquad\qquad t \uparrow (\theta \fatsemi \phi) = (t \uparrow \theta) \uparrow \phi$$

where $\mathsf{Term}\ n$ is the type of *well scoped* lambda-terms with n free variables in scope, and $n \sqsubseteq m$ is the type of order-preserving injections—*thinnings*—embedding n free variables into a larger scope with m free variables[1]. Moreover, with synthetic astonishment I can subsequently prove that $\cdot\uparrow\cdot$ respects thinning identity, ι, and composition, $\fatsemi$. In other words, I extend Term to a *presheaf* over (op-)thinnings — a functor into Type. And what is wrong is that *I* do it *myself*.

2 Worked Example

Let us take a closer look at the constructors of $\mathsf{Term}\ \cdot$ and the action of $\cdot\uparrow\cdot$:

$$\frac{x \,:\, 1 \sqsubseteq n}{\mathsf{var}\ x \,:\, \mathsf{Term}\ n} \quad (\mathsf{var}\ x) \uparrow \theta = \mathsf{var}\ (x \fatsemi \theta) \qquad \frac{f, s \,:\, \mathsf{Term}\ n}{\mathsf{app}\ f\ s \,:\, \mathsf{Term}\ n} \quad (\mathsf{app}\ f\ s) \uparrow \theta = \mathsf{app}\ (f \uparrow \theta)\ (s \uparrow \theta) \qquad \frac{t \,:\, \mathsf{Term}\ (n{+}1)}{\mathsf{lam}\ t \,:\, \mathsf{Term}\ n} \quad (\mathsf{lam}\ t) \uparrow = \mathsf{lam}\ (t \uparrow (\theta{+}1))$$

Note that for var, θ acts by *postcomposition*. Moreover, for lam, the postfix $\cdot{+}1$ which happens to n in the type looks a lot like the $\cdot{+}1$ which happens to θ in the function. What is the latter? Thinnings $n \sqsubseteq m$ (determining particular choices of n things from m) are generated thus:

$$\frac{}{0 \,:\, 0 \sqsubseteq 0} \qquad \frac{\theta \,:\, n \sqsubseteq m}{\theta\!\nearrow \,:\, n \sqsubseteq m{+}1} \qquad \frac{\theta \,:\, n \sqsubseteq m}{\theta{+}1 \,:\, n{+}1 \sqsubseteq m{+}1}$$

[1] I like m to be larger than n. Count the sticks.

The $\cdot{+}1$ action on a thinning coincides with the $\cdot{+}1$ action on source and target scopes. My overloading of the constructors is deliberate emphasis. The definitions of identity and composition confirm that $\cdot{+}1$ is a *functor*.

$$\begin{array}{rcl} \iota_0 & = & 0 \\ \iota_{(n+1)} & = & \iota_n{+}1 \end{array} \qquad \begin{array}{rcl} 0 \mathbin{;} 0 & = & 0 \\ \theta \mathbin{;} (\phi\curlywedge) & = & (\theta \mathbin{;} \phi)\curlywedge \\ (\theta\curlywedge) \mathbin{;} (\phi{+}1) & = & (\theta \mathbin{;} \phi)\curlywedge \\ (\theta{+}1) \mathbin{;} (\phi{+}1) & = & (\theta \mathbin{;} \phi){+}1 \end{array}$$

In this talk, we shall learn to spot such structure and make presheaves on purpose. We should be able to express the definition of Term in a way that points out how $1 \sqsubseteq \cdot$ is a functor from Thin to Type and $\cdot{+}1$ from Thin to Thin, recovering the action of Thin on Term automatically.

3 Prospectus

My specific plan is to construct a universe of datatype descriptions, as in previous work [4], by syntactifying a class of strictly positive functors whose least fixpoint may then be taken. But where we previously described indexed containers [2], we may now consider strictly positive functors between presheaves.

This construction necessarily relies on some formalisation of category theory, and it is a local non-goal for this to be a comprehensive treatment. We can get a long way with a notion of *small* categories and functors between them. The thinnings are exactly such a category, with $\cdot{+}1$ exactly such a functor.

We may then, separately, say what it is to be a presheaf — a functor from a small 'index' category into Type, and lift type constructors accordingly. In particular, the covariant Hom-functor (arrows from a given source) gives such a presheaf, with $1 \sqsubseteq \cdot$ a case in point. With this notion in place, we can give a syntax of descriptions for functors between presheaves, including the identity, constants, pairing, composition with a small functor, and pointwise Π and Σ over sets. When the source and target index categories of a description coincide, we may take a least fixpoint. We show once, for all such descriptions that this fixpoint a presheaf itself, obtaining the action of arrows in the index category and the proofs that identity and composition are respected.

4 Future Work

The idea that our indexed datatypes should respect more than discrete structure on indices is one small part of a broader agenda towards directed type theory [5, 3], but we can have it in our hands, now. Of course, we should very much like functor laws to hold *definitionally*, and that should be workable [1]. Let us see how much more categorical structure we can build into type theory for the price of having enough language to point it out.

References

[1] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In Stephanie Weirich, editor, *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*, pages 13–24. ACM, 2013.

[2] Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. Indexed containers. *J. Funct. Program.*, 25, 2015.

[3] Thorsten Altenkirch and Jacob Neumann. Synthetic 1-categories in directed type theory. *CoRR*, abs/2410.19520, 2024.

[4] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14. ACM, 2010.

[5] Daniel R. Licata and Robert Harper. 2-dimensional directed type theory. In Michael W. Mislove and Joël Ouaknine, editors, *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011*, volume 276 of *Electronic Notes in Theoretical Computer Science*, pages 263–289. Elsevier, 2011.

Thinning Thinnings: Safe and Efficient Binders

April Gonçalves[1] and Wen Kokke[2]

[1] University of Strathclyde, Glasgow, UK
[2] Well-Typed

Abstract

Binding is notoriously difficult to implement both correctly and efficiently. When using names, one must be ever vigilant for name capture. When using de Bruijn indices and thinnings, one must correctly adjust them whenever their scope changes. These invariants have tripped up even the most seasoned compiler writers. Using dependent types, it is straightforward to ensure the correct use of de Bruijn indices and thinnings. Unfortunately, such implementations are often inefficient. Moreover, this requires that the compiler is written in a dependently-typed language.

We present a Haskell library that provides an implementation de Bruijn indices and thinnings that is both correct and efficient. The library exports the usual inductive definitions for de Bruijn indices and thinnings, indexed on the type-level with the size of their scope and scopes, respectively. However, it implements these efficiently as machine words and bit vectors. We test the correctness of the efficient implementation using QuickCheck. We intend to benchmark the library against the usual inductive definitions, to ensure that it is more efficient, and against the unsafe efficient implementation, to ensure that the type-level indices do not incur any runtime overhead.

Background. Logicians, type theorists, and compiler writers have struggled with syntax and binding for over a century [3]. Compiler writers, especially, struggle with representing binding. *Named* representations of binding use matching names—be they strings or numbers—for binding sites and bound variables. Using names, one must work to avoid name capture and ensure α-equivalence. *Nameless* representations of binding got their party started with de Bruijn and his indices [1] which represent bound variables as numbers that index into the list of enclosing scopes, writing, e.g., $\lambda.(\lambda.1)$ for $\lambda x.(\lambda y.x)$. For the remainder of this abstract, we leave our stringly-named friends behind and focus on indices.

De Bruijn Indices and Thinnings, Inductively. A de Bruijn index selects an element from a list—be it a typing context or an evaluation environment, and we *can* represent them in Haskell as we will show... Everybody brace for length-indexed vectors! *Nat* is the kind of type-level natural numbers:

```
type data Nat = Z | S Nat   -- We write 0 for Z, 1 for S 0, 2 for S 1, 3 for S 2, etc.
```

Env n is the type of lists of length *n* and *Ix n* is the type of indices into such lists:

```
data Env n a where                          data Ix n where
  Nil ::                    Env Z     a       FZ ::          Ix (S n)
  Cons :: a → Env n a → Env (S n) a         FS :: Ix n → Ix (S n)
```

Where an index selects a single element from a list, a *thinning* selects a sublist. *Th m n* is the type of thinnings that thins a list of length *m* out to a list of length *n* by either keeping or dropping each element:

```
data Th m n where
  Done ::                Th Z       Z
  Keep :: Th m n → Th (S m) (S n)
  Drop :: Th m n → Th (S m)     n
```

```
ex1 :: Th 4 2   -- [A, B, C, D] to [C, D]
ex1 = Drop ∘ Drop ∘ Keep ∘ Keep $ Done
ex2 :: Th 4 2   -- [A, B, C, D] to [A, C]
ex2 = Keep ∘ Drop ∘ Keep ∘ Drop $ Done
```

Both *ex1* and *ex2* are examples of thinnings that select a sublist of length 2 from a list of length 4. Both have type *Th* 4 2. However, they are distinct: *ex1* drops elements 1 and 2 and *ex2* drops elements 2 and 4. The type *Th* 4 2 is inhabited by all combinations of selecting 2 elements from a list 4. This is such a general concept that thinnings, for compiler writers, pop up all over. In evaluation, a thinning can represent a batch of adjustments to the bound variable indices [5]. In dependent type checking, a thinning can represent the portion of its context to which a meta-variable has access. Using co-de Bruijn representation [5], thinnings can even be used to represent binding, obviating the need for indices.

De Bruijn Indices and Thinnings, Efficiently. The inductive definitions *Ix* and *Th* are correct but inefficient. The index 3, represented as *FS* (*FS FZ*), takes up *five* machine words. I hope we can agree that *one* machine word should suffice. If you *really* need more than 2^{64} nested binders[1], you can use Haskell's efficient arbitrary-precision *Integer* type[2]. Likewise, *ex1* and *ex2* each take up *nine* machine words, where 4 bits would suffice to represent each *Keep* and *Drop*. Such optimisations are often found in the wild. Both smalltt [4] and ask [6] represent indices as *Int* and thinnings as bit vectors—smalltt uses *Int*, which is efficient but limits the number of nested binders to 64, and ask uses *Integer* as an infinite bit vector. Unfortunately, these optimisations lose the type-level safety guarantees of the inductive definitions. Anecdotally, the most error-prone parts of both ask and Idris 1 [2] was in their implementations of binders.

Contribution. Our contribution is a library that provides an efficient implementation of de Bruijn indices and thinnings, representing indices as words and thinnings as bit vectors, but whose API matches the inductive definitions, including type-level safety guaratees. Our library leverages bidirectional pattern synonyms, view patterns, and—most importantly—*lies* to mimic the inductive data type definitions down to their constructors. Our technique leverages unsafe projection and embedding functions between the efficient representation and the inductive base functor, generating the inductive constructors on a by-need basis, e.g., indices are defined as:

```
data IxF f n where
  FZF ::          IxF f (S n)
  FSF :: f n → IxF f (S n)
```

```
newtype Ix (n :: Nat) = Ix Word16
unsafeProject :: Ix n          → IxF Ix n
unsafeEmbed   :: IxF Ix (S n) → Ix (S n)
```

```
pattern FZ   ← (unsafeProject → FZF)   where FZ   = unsafeEmbed FZF
pattern FS i ← (unsafeProject → FSF i) where FS i = unsafeEmbed (FSF i)
```

We test the equivalence of the inductive and the efficient definitions using a QuickCheck suite. In the future, we hope to benchmark our library against the inductive definitions. While the data type definitions make it sufficiently clear that our library is more space efficient, such a benchmark would test whether or not our library is more time efficient. Furthermore, we hope to benchmark our library against the efficient definitions *without* type-level safety guaratees, which would test whether or not our type-level indices introduce any runtime overhead.

[1] For simplicity, we assume a 64-bit architecture.

[2] But you should, probably, re-evaluate the decisions that led to you needing 18446744073709551615 variables.

References

[1] N. G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972.

[2] The Idris Developers. Idris 1. https://github.com/idris-lang/Idris-dev, 2025.

[3] Gottlob Frege. Uber sinn und bedeutung. *Zeitschrift für Philosophie Und Philosophische Kritik*, 100(1):25–50, 1892.

[4] András Kovács. smalltt. https://github.com/AndrasKovacs/smalltt, 2023.

[5] Conor McBride. Everybody's Got To Be Somewhere. *Electronic Proceedings in Theoretical Computer Science*, 275:53–69, July 2018.

[6] Conor McBride, Guillaume Allais, Fredrik Nordvall Forsberg, and Jules Hedges. ask. https://github.com/msp-strath/ask, 2025.

Categorical Normalization by Evaluation: A Novel Universal Property for Syntax

David G. Berry[1] and Marcelo P. Fiore[2*]

Computer Laboratory, University of Cambridge

[1] David.Berry@cl.cam.ac.uk
[2] Marcelo.Fiore@cl.cam.ac.uk

This work studies a categorical presentation of normalization by evaluation formalized in the Rocq proof assistant, resulting in an effectively executable algorithm for normalizing pure λ-calculus terms. Former work on categorical normalization, such as [1, 3, 4], has failed to establish both soundness and (strong) completeness of normalization by evaluation purely categorically. Either they have relied on a non-categorical proof, or they have relied on an *a priori* characterization of normal forms to deduce some properties. In this work we present a new universal property for syntax that supports the construction of a purely categorical argument for both soundness and (strong) completeness.

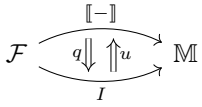
Categorical Normalization Normalization by Evaluation is a technique for computing the normal forms of λ-calculi terms in a reduction-free manner. It has seen many different expositions [2, 1, 3, 4]: proof-theoretic, type-theoretic, and category-theoretic. The category-theoretic presentation of Čubrić, Dybjer, and Scott by way of the Yoneda embedding [3] constructs a normal form algorithm purely functorially by the universal property of quotiented syntax within the setting of P-category theory. This ensures that the resultant algorithm satisfies soundness: that the output is $\beta\eta$-convertible with the input. Although, they resort to non-categorical techniques to establish that their normalization algorithm is (strongly) complete: that it maps $\beta\eta$-convertible inputs to α-equivalent outputs; and that their normalization algorithm produces β-normal–η-long–normal forms. The presentation of Fiore by way of categorical gluing [4] constructs a normal form algorithm using manually constructed maps over glued objects of neutral and normal forms. It is precisely because these forms are considered up to α-equivalence that the resultant algorithm is (strongly) complete.

Our approach combines techniques from these two prior works, with a new universal property to allow for a construction of a normal form algorithm that is both sound and (strongly) complete. This allows for an algorithm that reduces the problem of $\beta\eta$-convertibility to that of α-equivalence without needing an *a priori* known notion of normal form.

P-Category Theory Recently, E-category theory has received attention as a way of importing classical results into the setting provided by constructive proof assistants [5, 6]. Instead of using the in-built Martin-Löf Identity type for equality of morphisms, E-categories come equipped with their own equivalence relation on morphisms. This turns homs into setoids, requiring operations on morphisms to respect the equivalence relation. P-category theory [3] replaces the use of equivalence relations with partial equivalence relations (*i.e.* relaxing reflexivity), thereby resulting in hom-subsetoids. This subtle change comes with technical advantages that provide a more appropriate setting for categorical normalization algorithms. We continue in Čubrić, Dybjer, and Scott's [3] use of P-category theory to phrase our construction and formalization.

*Research partially supported by EPSRC grant EP/V002309/1.

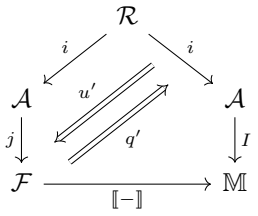
Universal Properties of Syntax The free Cartesian-closed category[1], $\mathcal{F}$, satisfies an initiality property.



For any pointed Cartesian-closed category, $\mathbb{M}$, and strictly-pointed Cartesian-closed functor, I, thereinto, there is a universal strictly-pointed Cartesian-closed interpretation functor, $[\![-]\!]$, from the free Cartesian-closed category, $\mathcal{F}$, resulting in a unique natural isomorphism, $q : [\![-]\!] \mathrel{\tilde{\Rightarrow}} I : u$.

Čubrić, Dybjer, and Scott [3] use this fact to deduce their normal form algorithm. Moreover, their switch to P-category theory allows them to consider a second category: the category of unquotiented (*i.e.* up to α-equivalence) terms. They remark that this category is initial over some term algebra but seemingly make no use of this fact. They use presheaves over this category to deduce *non-categorically* that their algorithm is (strongly) complete. Their use of P-category theory allows them to conclude that the computational aspect of presheaves over the two categories are identical. Indeed it was a desire to replicate their argument more categorically that resulted in the presented work.

The category of unquotiented simply-typed λ-calculus terms, $\mathcal{A}$, also satisfies an initiality property but with respect to the free Cartesian-closed category, $\mathcal{F}$.



For any pointed Cartesian-closed category, $\mathbb{M}$, and strictly-pointed Cartesian-pre-closed functor, I, thereinto, there is a universal strictly-pointed Cartesian-closed interpretation functor, $[\![-]\!]$, from the free Cartesian-closed category, $\mathcal{F}$, resulting in a unique pair of natural transformations, $q' : [\![-]\!] \circ j \circ i \Rightarrow I \circ i$, and u' in the reverse direction. Where the functor j is the quotient map from unquotiented syntax to quotiented syntax, the category $\mathcal{R}$ is the category of renamings, and i is the inclusion map from renamings into (unquotiented) syntax. And Cartesian-pre-closure is a novel condition weaker than full Cartesian-closure for functors that arises from the weaker structure of $\mathcal{A}$. A category, $\mathbb{C}$, is Cartesian-pre-closed precisely when:

- it is Cartesian;
- it has a pre-exponential operator on objects:
$$(-) \Rightarrow (=) : \mathbb{C}_0 \times \mathbb{C}_0 \to \mathbb{C}_0;$$
- such that there are maps natural in c:
$$\mathbb{C}(c \times a, b) \Rightarrow \mathbb{C}(c, a \Rightarrow b)$$
$$\mathbb{C}(c, a \Rightarrow b) \Rightarrow \mathbb{C}(c \times a, b).$$

A functor, $F : \mathbb{C} \to \mathbb{D}$, with $\mathbb{C}$ Cartesian-pre-closed and $\mathbb{D}$ Cartesian-closed, is Cartesian-pre-closed precisely when:

[1] We consider freeness over a single base type in our work; one may easily instead consider freeness over a set of base types.

- it is Cartesian;
- there is a family of maps:

$$e' : F(a) \Rightarrow F(b) \to F(a \Rightarrow b);$$

- such that the following holds:

$$e' \circ (F(f) \circ p)^* \sim F(f^*).$$

By judiciously instantiating $\mathbb{M}$ and I with presheaves over $\mathcal{A}$ and the Yoneda embedding one can replicate the construction, *mutatis mutandis*, from [3] and induce a normal form algorithm that is strongly complete purely categorically. However this algorithm is not obviously sound due to the weaker naturality properties of q' and u'.

This shortfall can be overcome by further judicious instantiation of $\mathbb{M}$ with a gluing category allowing for an algorithm that is both sound and strongly complete. There are a number of possible candidates for such a gluing category that allow for the desired conclusions of soundness and (strong) completeness, each with their own resultant computational aspects. It suffices for the result to use the arrow category of presheaves over $\mathcal{A}$ for $\mathbb{M}$, and the pairing of the Yoneda embedding with the relative embedding along j for I. This realises the domain and codomain parts, respectively, as containing strict information for (strong) completeness, and weak quotient information for soundness. This gluing construction can be seen to combine aspects of [4] with our new universal property permitting the induction of a normal form algorithm purely categorically without any *a priori* knowledge of neutral and normal forms.

References

[1] T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science*, pages 182–199, Berlin, Heidelberg, 1995. Springer.

[2] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, 1991.

[3] D. Čubrić, P. Dybjer, and P. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8(2):153–192, 1998.

[4] M. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. *Mathematical Structures in Computer Science*, 32(8):1028–1065, 2022.

[5] J. Gross, A. Chlipala, and D. I. Spivak. Experience implementing a performant category-theory library in Coq. In *Interactive Theorem Proving*, pages 275–291. Springer, 2014.

[6] J. Z. S. Hu and J. Carette. Formalizing category theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 2021.

Rational Codata as Syntax-with-Binding:
Correct-by-Construction Foundations of the Modal μ-Calculus*

Sean Watters

University of Strathclyde, UK
sean.watters@strath.ac.uk

The modal μ-calculus is an extension of the basic modal logic K with least and greatest fixpoint operators. It is of foundational importance in the field of model checking [1], and also of considerable theoretical interest to logicians for its rich properties. The model checking and satisfiability problems for the μ-calculus are decidable, yet the logic is highly expressive — it subsumes the temporal logics LTL, CTL, CTL* and propositional dynamic logic, and is equivalent to the bisimulation-invariant fragment of monadic second-order logic [5]. Of interest to the practising type theorist, the μ-calculus admits a constructive semantics in containers which is currently in the early stages of being explored [10].

This (still work-in-progress) project is concerned with formalising the correctness and finiteness proofs for a particular key syntactic construction on μ-calculus formulas, namely their (Fischer-Ladner) *closure*. This can be thought of as a succinct graph representation of the formula. The definition of the closure is not structurally recursive, and indeed, the proof of its finiteness (originally due to Kozen [6]) is fairly technical, particularly in a formal setting where issues like variable binding and substitution must be treated seriously.

A key feature of our work is the use of syntax-with-binding as an inductive presentation of rational codata — data structures that loop back on themselves — in a manner similar to the technique first demonstrated by Ghani et al [3], and Hamana [4].

We work in Agda, using guarded corecursion. At the time of submission, the bulk of the formalisation is complete (up to and including Theorem 1). Theorem 2 is still a work-in-progress.

The μ-Calculus Formulas in the modal μ-calculus are generated by the following grammar, parameterised by some set A of atomic propositions. We represent formulas in Agda as an inductive data type using well-scoped de Bruijn indices for the variables, but for convenience we write variable names here. Note the lack of implication or arbitrary negations; this is required to maintain strict positivity, which is important when giving the semantics of the fixpoint operators.

$$\mu\mathsf{ML} := \top \mid \bot \mid A \mid \neg A \mid \mu\mathsf{ML} \wedge \mu\mathsf{ML} \mid \mu\mathsf{ML} \vee \mu\mathsf{ML} \mid \Box\, \mu\mathsf{ML} \mid \Diamond\, \mu\mathsf{ML} \mid \mu x.\mu\mathsf{ML} \mid \nu x.\mu\mathsf{ML}$$

We can define substitution for these formulas in standard fashion, which allows us to define the *unfolding* of a fixpoint formula as its direct subformula with the outermost variable substituted for the whole formula: $\mathsf{unfold}\ (\mu x.\varphi) = \varphi\ [x := \mu x.\varphi]$.

The Closure The closure of a μ-calculus formula φ is the least set of formulas that contains φ and is closed under the "single-step closure relation", which relates modal and propositional formulas to their direct subformulas, and fixpoint formulas to their unfoldings. We define the membership relation for the closure $- \in_C -$ as the transitive reflexive closure of this single-step relation, where "$\psi \in_C \varphi$" means "ψ is in the closure of φ".

*Work-in-progress Agda formalisation available at https://github.com/Sean-Watters/mu-calc-de-bruijn

The membership relation $\in_C$ allows us to characterise the closure, but does not provide an obviously terminating algorithm to compute it. Note that it is not immediate from the definition of $\in_C$ that the closure graph only contains finitely many nodes, as the unfolding of φ is not structurally smaller than φ, and it is not immediately clear that a chain of unfoldings would eventually loop back on itself.

Kozen's original proof of finiteness involves an alternative definition of the closure which is structurally recursive. The key to it is the so-called *expansion map*, which can be thought of as the "maximal unfolding" — the expansion map sequentially instantiates every free variable for its binder, in order from most recently bound to least. This definition of the closure is finite by construction, but it is difficult to directly prove it equivalent to the standard definition in a formal setting.

Our approach is to first define two closure algorithms; one that produces an infinite cotree via guarded corecursion directly from the definition of $\in_C$ (which is trivially correct), and one via Kozen's inductive algorithm (which is intrisically finite). Crucially, the inductive data structure we use for the latter includes a notion of *back-edges*, which allows us to unfold it to an infinite codata structure. We then prove that the coinductive algorithm is bisimilar to the unfolding of the inductive one, and by doing so, obtain correctness and finiteness for both constructions.

Rational Codata Just as a rational number is one whose decimal expansion is finitely presentable as a prefix plus a loop, an element of a rational codata type is an infinite data structure for which every infinite path eventually falls into a repeating cycle. They arise via the *rational* fixpoint of endofuctors, which is a particular sub-coalgebra of the final coalgebra [8][9].

We argue that the correct way to think about the closure of a μ-calculus formula is as a rational codata structure. Inductive structures such as lists and trees forget the backedges, which are crucial to the properties we wish to prove, meanwhile simply using cotrees forgets any evidence we may have had about the closure being finitely presentable.

In the style of Ghani et al. [3], we represent the closure as an inductive "μ-calculus formula-shaped tree". That is: such trees store data at both nodes and leaves, nodes are labelled by formula constructors $\{\wedge, \vee, \Box, \Diamond, \mu, \nu\}$ (and have the appropriate arities), and we also have variables which we consider as pointers to $\{\mu, \nu\}$-nodes. Thus, this presentation is a syntax-with-binding. We define their unfolding to non-wellfounded cotrees (with the same node arities and labelling) via guarded corecursion.

Bisimilarity for cotrees is defined as the pointwise lifting of propositional equality, for which we create a DSL for compositional proof, following the technique first demonstrated by Danielsson [2]. We now have all we need to prove that the two definitions of the closure are equivalent:

Theorem 1. *The direct coinductive definition of the closure is bisimilar to the unfolding of the inductive syntax-with-binding definition.*

Our notions of correctness for each algorithm relate the paths through the tree produced to $\in_C$, in the sense that a path exists from φ to ψ iff $\psi \in_C \varphi$. For both kinds of tree, this is an instance of the "Any" predicate transformer (or rather, in the inductive case, a variant of it that allows backedge traversal). If we can transport proofs of Any across bisimulations of the form found in Theorem 1, then we can prove the finite-by-construction inductive algorithm correct. The proof of this fact is still work-in-progress.

Theorem 2 (WIP)**.** *Let T be the tree with backedges produced by the inductive closure algorithm applied to φ. Then for all formulas ψ there is a path to ψ in T iff $\psi \in_C \varphi$. That is, T really is the closure of φ.*

Conclusions and Future Work We believe this to be the first serious attempt at formalising the Fischer-Ladner closure of the modal μ-calculus in a proof assistant, though others have previously formalised different aspects of the μ-calculus in Agda [10], Rocq [7], and LEGO [11]. A natural next step in this programme would be to formalise the game semantics of the μ-calculus (where the closure plays a key role), and in doing so obtain a correct-by-construction model checker.

In our opinion, the modal μ-calculus and type theory are ripe for further cross-fertilisation. For example, we believe that rational codata types are currently under-explored, and that this work provides a good example of their utility. In particular, we would like to further investigate the inductive syntax-with-binding presentation and its unfolding, and whether these generalise usefully to containers. Our hope is to develop a type-theoretic framework for ergonomically reasoning about finitely-presentable infinite data.

References

[1] Julian Bradfield and Igor Walukiewicz. *The mu-calculus and Model Checking*, pages 871–919. Springer International Publishing, Cham, 2018.

[2] Nils Anders Danielsson. Beating the productivity checker using embedded languages. *arXiv preprint arXiv:1012.4898*, 2010.

[3] Neil Ghani, Makoto Hamana, Tarmo Uustalu, and Varmo Vene. Representing cyclic structures as nested datatypes. In *Proc. of 7th Symp. on Trends in Functional Programming, TFP*, volume 2006, 2006.

[4] Makoto Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6, 2010.

[5] David Janin and Igor Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *International Conference on Concurrency Theory*, pages 263–277. Springer, 1996.

[6] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3):333–354, 1983.

[7] Marino Miculan. On the formalization of the modal µ-calculus in the calculus of inductive constructions. *Inf. Comput.*, 164:199–231, 2001.

[8] Stefan Milius. A sound and complete calculus for finite stream circuits. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 421–430. IEEE, 2010.

[9] Stefan Milius. Proper functors and their rational fixed point. In *7th Conference on Algebra and Coalgebra in Computer Science (CALCO 2017)*, pages 18–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

[10] Ivan Todorov and Casper Bach Poulsen. Modal μ-calculus for free in agda. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development*, pages 16–28, 2024.

[11] Shenwei Yu and Zhaohui Luo. Implementing a model checker for lego. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 442–458, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

Dependent two-sided fibrations for directed type theory

Fernando Chu and Paige Randall North

Utrecht University, Netherlands

Background

In a seminal paper, Hofmann and Streicher [1] constructed the *groupoid model* for Martin-Löf Type Theory (MLTT). In this model, a type A is interpreted as a groupoid $[\![A]\!]$, while its identity type Id_A is interpreted as the hom-set of $[\![A]\!]$. Further work has shown that types can be also interpreted as ∞-groupoids [2], and that, together with the univalence axiom, the resulting theory allows for a synthetic development of ∞-groupoids [3].

More recently, new, *directed*, type theories have emerged, with the goal of developing a way of doing synthetic category theory. Different approaches have been used, such as a 2-dimensional theories [4], modal typings [5, 6, 7], or multilayered approaches [8], among others. Arguably the most successful of these are Simplicial Type Theory [8] and Triangulated Type Theory [6]. However, they achieve their expressive power by interpreting types not as categories, but more general structures, and then carving out those types that behave like categories.

Our work aims to develop a type theory with comparable expressive power to the aforementioned ones, but without changing the universe of types. It builds on previous joint work with Mangel [9], which itself is a continuation of the work done in [10]. The key new contributions are a new context extension rule and modified rules for hom-types, which we now sketch.

An overview of directed type theory

We sketch some basic components in our theory, which were already present in [10]. Extending the groupoid model, we define contexts to be categories, with the empty context being the terminal category. A type in a context Γ is a functor $A : \Gamma \to \mathsf{Cat}$, it follows that a type in the empty context is a category. The context extension operation $\Gamma \rhd A$ is interpreted as the Grothendieck construction $\int_\Gamma A$. As usual, terms of A are sections of the canonical projection $\Gamma \rhd A \to \Gamma$.

Furthermore, we have types A^{core} and A^{op} for each type A, which in the semantics are obtained by postcomposing $A : \Gamma \to \mathsf{Cat}$ with the endofunctors $\mathsf{core}, \mathsf{op} : \mathsf{Cat} \to \mathsf{Cat}$, respectively. We additionally have a hom-type former, with formation rule:

$$\text{hom-}\textsc{Form}\ \frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A^{\mathsf{op}}, y : A \vdash \hom_A(x, y)\ \mathsf{type}}$$

In the empty context, this type is interpreted as the usual functor $\hom_A : A^{\mathsf{op}} \times A \to \mathsf{Set}$, except that each set is interpreted as a discrete category.

The dependent 2-sided context extension

In addition to the usual context extension rule, we now have a "dependent 2-sided" version:

$$\frac{\Gamma\,\mathsf{ctx} \qquad \Gamma \vdash A\ \mathsf{type} \qquad \Gamma, a : A^{\mathsf{op}} \vdash B(a)\ \mathsf{type}}{\Gamma, a : A, b \overset{2\mathsf{f}}{:} B(\bar{a})\,\mathsf{ctx}}\ \textsc{Ctx-Ext}_2$$

Semantically, it corresponds to a dependent analogue of Street's notion of a 2-sided fibration [11], which we call a **dependent 2-sided fibration (D2SFib)**. This construction is equipped with an opfibration over Γ and a "local fibration" over $\int_\Gamma A$, and is the universal such construction in the sense that we obtain an equivalence of categories

$$\mathsf{Functor}(\int_\Gamma (\mathsf{op} \circ A), \mathsf{Cat}) \simeq \mathsf{D2SFib}(\Gamma, A)$$

We now interpret terms $\Gamma, a : A \vdash b \overset{2\mathsf{f}}{:} B$ as sections $(\Gamma, a : A) \to (\Gamma, a : A, b \overset{2\mathsf{f}}{:} B)$.

However, D2SFibs are not stable under pullback, which requires us to make a new judgment to capture all substitutions. We write $\Gamma \vdash X\ \mathsf{type}_{\mathfrak{d}}$ for any displayed category $X \to \Gamma$; equivalently, for a normal lax functor X from Γ to the double category of profunctors Prof. Now, terms in X correspond to sections of the displayed category, and so we can write:

$$\frac{\Gamma, x : X, \Delta \vdash a \overset{2\mathsf{f}}{:} X \qquad \Gamma \vdash m : X}{\Gamma, \Delta[m/x] \vdash a \overset{\mathfrak{d}}{:} X}\ \textsc{Subst}$$

New rules for the hom-type

In particular, when applying this new operation to the hom-type we are able to form the new context $b : A, \bar{a} : A, f \overset{2\mathsf{f}}{:} \hom_A(a, b)\,\mathsf{ctx}$ (note the switching of the variables). Semantically, this corresponds to the arrow category $A^{\to}$, which validates the following rules:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash x : A}{\Gamma \vdash \mathsf{refl}_x \overset{\mathfrak{d}}{:} \hom(\bar{x}, x)}\ \textsc{hom-Intro}$$

$$\frac{\begin{array}{c}\Gamma, a : A \vdash X\ \mathsf{type} \\ \Gamma, b : A, a : A, f \overset{2\mathsf{f}}{:} \hom_A(\bar{a}, b), x : X^{\mathsf{op}} \vdash D\ \mathsf{type} \\ \Gamma, a : A, x : X \vdash d \overset{\mathfrak{d}}{:} D[a/b, \mathsf{refl}_A/f]\end{array}}{\Gamma, b : A, a : A, f \overset{2\mathsf{f}}{:} \hom_A(\bar{a}, b), x : X \vdash j_d \overset{2\mathsf{f}}{:} D}\ \textsc{hom-Elim}$$

As an example of the semantics, the hom introduction rule in the empty context states that the diagonal $\Delta : A \to A \times A$ lifts along the projection $A^{\to} \to A$. For another example, we can now precisely capture natural transformations between two functors $F, G : A \to B$, they correspond to judgements $a : A \vdash \tau_a : \hom(F\bar{a}, Ga)$.

Furthermore, these rules plus the directed analogue of function extensionality are sufficient to develop some category theory. Indeed, we can show:

Lemma (Yoneda)**.** *Let $A : \mathcal{U}^{\mathsf{core}}$ be a type. Then, the following two functors are naturally*

isomorphic

$$\begin{aligned} Y, \bar{Y} &: (A \to \mathsf{Set}) \times A \to \mathsf{Set} \\ Y(F, a) &:\equiv \prod_{(x,f):\sum_{x:A} \hom(a,x)} Fx \\ \bar{Y}(F, a) &:\equiv Fa \end{aligned}$$

Future works

This is still work in progress; in particular, we need to better understand D2SFibs, their pullbacks, and their relation with factorization systems. Additionally, we still have to investigate just how much category theory can be developed internally. We hope that properties of Triangulated Type Theory, such as a directed structure identity principle, have an analog for our syntax.

References

[1] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-Five Years of Constructive Type Theory*, pages 83–111. Oxford University Press, 1998.

[2] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of Univalent Foundations (after Voevodsky). *Journal of the European Mathematical Society*, 23(6):2071–2126, March 2021.

[3] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[4] Daniel R. Licata and Robert Harper. 2-Dimensional Directed Type Theory. *Electronic Notes in Theoretical Computer Science*, 276:263–289, September 2011.

[5] Nuyts Andreas. Towards a Directed Homotopy Type Theory based on 4 Kinds of Variance. Master's thesis, KU Leuven, 2015.

[6] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. Directed univalence in simplicial homotopy type theory, 2024. Version Number: 1.

[7] Thorsten Altenkirch and Jacob Neumann. Synthetic 1-Categories in Directed Type Theory, 2024. Version Number: 1.

[8] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories. *Higher Structures*, 1(1):147–224, December 2017.

[9] Chu, Fernando, Mangel, Éléonore, and North, Paige Randall. A directed type theory for 1-categories. In *30th International Conference on Types for Proofs and Programs (TYPES 2024)*, 2024.

[10] Paige Randall North. Towards a Directed Homotopy Type Theory. *Electronic Notes in Theoretical Computer Science*, 347:223–239, November 2019.

[11] Ross Street. Fibrations in bicategories. *Cahiers de topologie et géométrie différentielle*, 21(2):111–160, 1980. (Corrections in 28(1):53–56, 1987).

Synthetic-Inductive Category Theory

Jacob Neumann[1,2]

[1] University of Nottingham, Nottingham, United Kingdom
jacob.neumann@nottingham.ac.uk
[2] Reykjavik University, Reykjavik, Iceland

One key insight emerging from the development of Martin-Löf Type Theory is that *types are synthetic groupoids*. That is, the identity types of a given type A possess the structure of a groupoid—the terms are the objects, refl_t is the identity morphism on $t\colon A$, the invertibility of identity terms encodes the fact that every morphism is an isomorphism, and so on. The "lower-dimensional" version of this observation was made in the 1990s, most famously in Hofmann and Streicher's *groupoid model* [3]. In the groupoid model, the types are synthetic 1-groupoids: the model's refutation of the *uniqueness of identity proofs* corresponds to the fact that there may be multiple distinct morphisms between any two objects in a groupoid, but the identity structure becomes propositional beyond that point. Using type theory as a synthetic language for *higher* groupoids, indeed ∞-*groupoids*, was a key impetus behind the development of *homotopy type theory* [10]. In homotopy type theory, the types are natively higher groupoids [11], with the identity types possessing (potentially) nontrivial identity type structure of their own, and those identity types having interesting identity types, and so on.

These developments led naturally to the question: what about synthetic *categories*? That is, can the symmetry of identity types be dropped (yielding "hom-types"), so that our types are instead natively endowed with the structure of synthetic *categories*? This question proves to be more challenging, but the outlook of such **directed type theory** is quite hopeful. Approaches to directed type theory also come in higher- and lower-dimensional flavours. In recent years, there has been a significant amount of interest in the *simplicial* approach to synthetic ∞-category theory pioneered by Riehl and Shulman[9], with important pieces of ∞-category theory being performed in the synthetic style [12, 1, 2] and an experimental proof assistant RZK being developed to allow for computer formalisation [4]. The lower-dimensional track—a "directed analogue" of Hofmann-Streicher—is less-developed: though the appropriate definition of the *category model* and its hom-types are known [8], there has yet to be a consensus on how to arrange the category model's *polarity calculus* in such a way to make **synthetic 1-category theory** possible. In recent work [7, 6], I[1] propose a resolution to these issues (combining aspects of the type theories of North [8] and Licata-Harper [5], plus further innovations), and show how the category model interprets a directed type theory capable of synthetic 1-category theory.

The purpose of this talk is to expound this directed type theory, and show the style of synthetic category theory it permits. Following the HoTT Book [10], we adopt an informal style more suited to mathematical practice. In this style of directed type theory, a modal typing discipline is adopted to track the *variances* of terms: $t\colon A$ means that t is *covariant*, whereas $t\colon A^-$ means that t is *contravariant*. Viewing types as synthetic categories, this is the same as saying that A^- is the *opposite category* of A. Accordingly, we type our hom-type formation rule as follows.

$$x\colon A^-, y\colon A \vdash \mathrm{hom}(x,y)\ \mathrm{type}$$

These polarities have a *substructural* character: for *closed* terms we are able to freely coerce between A^- and A (for every closed term $t\colon A^-$, there is a closed term $-t\colon A$ and *vice-versa*)

[1] Jointly with Thorsten Altenkirch.

but for *open* terms we have no such operation. Therefore, the reflexivity term—the identity morphisms in our synthetic category theory—are stated only for closed terms.

$$\frac{t\colon A^-}{\mathsf{refl}_t\colon \hom(t,-t)}$$

The purpose of these restrictions is to prevent the elimination principle for hom-types—the *directed J-rule*—from being able to prove symmetry. We can prove by metatheoretic argument that our directed J-rule,

$$\frac{\begin{array}{c} t\colon A^- \\ y\colon A, u\colon \hom(t,y) \vdash M(y,u)\ \mathrm{type} \\ m\colon M(-t,\mathsf{refl}_t) \end{array}}{y\colon A, u\colon \hom(t,y) \vdash J_{t,M}(m)\colon M(y,u)}$$

cannot prove symmetry: it is validated by the category model, but the category model refutes symmetry (not every category is a groupoid). Using this principle of directed path induction, we can easily construct the composition of hom-terms [2] and prove it satisfies the category laws.

When we carry out synthetic category theory in this setting, it permits a novel style of category theory, which we call **inductive category theory**. In this framework, the *universal mapping properties* of traditional category theory are instead reframed as *principles of induction*. For instance, we make the following definition: given terms $s, t\colon A$, we say that a term $s \times t\colon A^-$ and terms $\pi_1\colon \hom(s \times t, s)$ and $\pi_2\colon \hom(s \times t, t)$ constitute a *product* of s and t if it satisfies the following principle of induction.

$$\frac{\begin{array}{c} z\colon A^-, u\colon \hom(z,s), v\colon \hom(z,t) \vdash M(z,u,v)\ \mathrm{type} \\ m\colon M(s \times t, \pi_1, \pi_2) \end{array}}{z\colon A^-, u\colon \hom(z,s), v\colon \hom(z,t) \vdash \mathsf{elim}(m)\colon M(z,u,v)}$$

From here, we can recover the usual universal mapping property. For instance, given a *cone* $k\colon A^-$, $f\colon \hom(k,s)$, $g\colon \hom(k,t)$, we can obtain the hom-term $\langle f, g\rangle\colon \hom(k, -(s \times t))$ by induction: just set $\langle \pi_1, \pi_2\rangle$ equal to $\mathsf{refl}_{s\times t}$ and we're done.

In this talk, we'll explore this peculiar method of phrasing category-theoretic concepts, showing that it can encompass more complex notions like exponentials and adjoints, and speculate on what a fully-worked out category theory in this style might look like.

References

[1] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. Directed univalence in simplicial homotopy type theory, 2024.

[2] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. The yoneda embedding in simplicial type theory, 2025.

[3] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. *Twenty-five years of constructive type theory (Venice, 1995)*, 36:83–111, 1995.

[4] Nikolai Kudasov, Emily Riehl, and Jonathan Weinberger. Formalizing the ∞-categorical yoneda lemma, 2023.

[5] Daniel R Licata and Robert Harper. 2-dimensional directed dependent type theory. 2011.

[2] For any $f\colon \hom(t,t')$, it suffices to define $f \cdot \mathsf{refl}_{-t'} = f$ to define the $f \cdot \_$ operation sending any $g\colon \hom(-t', t'')$ to $f \cdot g\colon \hom(t, t'')$.

[6] Jacob Neumann. *A Generalized Algebraic Theory of Directed Equality*. PhD thesis, University of Nottingham, 2025.

[7] Jacob Neumann and Thorsten Altenkirch. Synthetic 1-categories in directed type theory. *arXiv preprint arXiv:2410.19520*, 2024.

[8] Paige Randall North. Towards a directed homotopy type theory. *Electronic Notes in Theoretical Computer Science*, 347:223–239, 2019.

[9] E. Riehl and M. Shulman. A type theory for synthetic ∞-categories. *Higher Structures*, 1(1):116–193, 2017.

[10] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[11] Benno Van Den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the london mathematical society*, 102(2):370–394, 2011.

[12] Jonathan Weinberger. A synthetic perspective on (∞,1)-category theory: Fibrational and semantic aspects. 2022.

Directed equality with dinaturality

Andrea Laretto, Fosco Loregian, and Niccolò Veltri*

Tallinn University of Technology, Tallinn, Estonia

Equality in Martin-Löf type theory is inherently symmetric [8]: this is what allows for types to be interpreted as sets, groupoids [9], and ∞-groupoids [19]; points of a type correspond to objects, and equality is precisely interpreted by morphisms which are always invertible.

A natural question follows: can there be a variant of Martin-Löf type theory which enables types to be interpreted as *categories*, where morphisms need not be invertible? Such a system should take the name of *directed type theory* [13, 16, 1, 6, 11, 2] (DTT), where the directed aspect comes from a non-symmetric interpretation of "equality", which now has a source and a target in the same way that morphisms do in a category. A common feature of current semantic approaches to directed type theory is to resort back to the maximal sub*groupoid* $\mathbb{C}^{\mathsf{core}}$ of categories [16, 2] in order to use the same variable with different variances $\mathbb{C}$ and $\mathbb{C}^{\mathsf{op}}$; this is needed to validate introduction (refl) and elimination (J) rules for directed equality.

Dinaturality for directed type theory. In this work, we describe a first-order non-dependent proof-relevant type theory where types are interpreted as categories, terms as functors, predicates as endoprofunctors and entailments as *dinatural transformations* [4]; intuitively, dinatural transformations allow for the same variable to appear both covariantly and contravariantly, and terms are required to be given only "on the diagonal" by equating the two occurrences with the same value, thus avoiding the need for groupoids. An excerpt of the rules of our type theory is shown in Figure 1, where (refl) and (J) capture the rules for directed equality. This type theory is equipped with a syntactic notion of polarity which allows variables to be distinguished based on their appearance in negative and positive positions, which are represented in entailments as $\overline{x} : \mathbb{C}^{\mathsf{op}}$ and $x : \mathbb{C}$, respectively. Such polarity is used to express a syntactic requirement on the J-rule: given a directed equality in context $\hom_{\mathbb{C}}(x, y)$ with $x : \mathbb{C}^{\mathsf{op}}, y : \mathbb{C}$, then x and y are allowed to be contracted to the same variable z only if both x and y appear only positively (i.e., with the same polarity) in the conclusion and only negatively (i.e., with the opposite polarity) in the context. This rule allows us to syntactically recover the same definitions about equality that one expects in standard Martin-Löf type theory, *except for symmetry* because of the syntactic restrictions: e.g., transitivity of directed equality (the composition map in a category), congruences of terms along directed equalities (the action of a functor on morphisms), transport along directed equalities (the coYoneda lemma), and an internal (di)naturality statement. Proving equational properties about these maps also follows the same steps as in Martin-Löf type theory using *directed equality induction*, given in (J-eq), which is a "dependent" version of (J) for the judgement of equalities of entailments. Crucially, (J-eq) is validated in the model using dinaturality of maps. As in the case of symmetric equality [10, Lemma 3.2.3], (J) is actually an isomorphism, and the inverse map is given by precomposing with (refl). Finally, the interval type $\mathbf{I} := \{0 \to 1\}$ with a single arrow serves as countermodel for symmetry of directed equality.

Dinaturals famously do not compose [18]; the practical consequence of this fact is that in this type theory there is no general cut rule for entailments. We provide two restricted cut rules (cut-din) and (cut-nat), intuitively capturing the composition of dinaturals with *naturals*, which are enough to capture all practical cases in which composition is needed. The rule (cut-assoc) captures associativity of these two compositions in the equational theory for entailments.

*Loregian was supported by the Estonian Research Council grant PRG1210. Veltri was supported by the Estonian Research Council grant PSG749.

$$\boxed{[\Gamma]\ \Phi \vdash \alpha : P} \qquad \frac{}{[\Gamma]\ \Phi, a : P, \Phi' \vdash a : P}\ (\mathsf{var}) \qquad \frac{[\Gamma]\ \Phi \vdash \alpha : P}{[\Gamma]\ A, \Phi \vdash \mathsf{wk}(\alpha) : P}\ (\mathsf{wk}) \qquad \frac{}{[\Gamma]\ \Phi \vdash ! : \top}\ (\top)$$

$$\frac{[x : \mathbb{C}, \Gamma]\ \Phi(\overline{x}, x) \vdash \alpha : P(\overline{x}, x)}{[x : \mathbb{C}^{\mathsf{op}}, \Gamma]\ \Phi(x, \overline{x}) \vdash \alpha : P(x, \overline{x})}\ (\mathsf{op}) \qquad \frac{\Gamma \vdash F : \mathbb{C} \qquad [x : \mathbb{C}, \Gamma]\ \Phi(\overline{x}, x) \vdash \alpha : Q(\overline{x}, x)}{[\Gamma]\ \Phi(F^{\mathsf{op}}(\overline{x}), F(x)) \vdash F^*(\alpha) : Q(F^{\mathsf{op}}(\overline{x}), F(x))}\ (\mathsf{idx})$$

$$\frac{[\Gamma]\ \Phi \vdash P \times Q}{[\Gamma]\ \Phi \vdash P, \qquad [\Gamma]\ \Phi \vdash Q}\ (\mathsf{prod}) \qquad \frac{[x : \Gamma]\ A(\overline{x}, x), \Phi(\overline{x}, x) \vdash B(\overline{x}, x)}{[x : \Gamma] \qquad \Phi(\overline{x}, x) \vdash A^{\mathsf{op}}(x, \overline{x}) \Rightarrow B(\overline{x}, x)}\ (\mathsf{exp})$$

$$\frac{\begin{array}{r}[a : \Delta^{\mathsf{op}}, b : \Delta, x : \Gamma]\ \Phi(\overline{x}, x, a, b) \vdash \alpha \quad : P(a, b) \\ [z : \Delta, x : \Gamma]\ k : P(\overline{z}, z), \Phi(\overline{x}, x, \overline{z}, z) \vdash \gamma[k] \ : Q(\overline{z}, z)\end{array}}{[z : \Delta, x : \Gamma]\ \Phi(\overline{x}, x, \overline{z}, z) \vdash \gamma[\alpha] : Q(\overline{z}, z)}\ (\mathsf{cut\text{-}din)}$$

$$\frac{\begin{array}{r}[z : \Delta, x : \Gamma]\ \Phi(\overline{x}, x, \overline{z}, z) \vdash \gamma \quad : P(\overline{z}, z) \\ [a : \Delta^{\mathsf{op}}, b : \Delta, x : \Gamma]\ k : P(a, b), \Phi(\overline{x}, x, \overline{a}, \overline{b}) \vdash \alpha[k] : Q(a, b)\end{array}}{[z : \Delta, x : \Gamma]\ \Phi(\overline{x}, x, \overline{z}, z) \vdash \alpha[\gamma] : Q(\overline{z}, z)}\ (\mathsf{cut\text{-}nat})$$

$$\frac{}{[x : \mathbb{C}, \Gamma]\ \Phi \vdash \mathsf{refl}_{\mathbb{C}} : \hom_{\mathbb{C}}(\overline{x}, x)}\ (\mathsf{refl}) \qquad \frac{[z : \mathbb{C}, \Gamma] \qquad \Phi(\overline{z}, z) \vdash h : P(\overline{z}, z)}{[a : \mathbb{C}^{\mathsf{op}}, b : \mathbb{C}, \Gamma]\ e : \hom_{\mathbb{C}}(a, b), \Phi(\overline{b}, \overline{a}) \vdash J(h)[e] : P(a, b)}\ (J)$$

$$\frac{[a : \mathbb{C}, \Gamma]\ \Phi \vdash Q(\overline{a}, a)}{[\Gamma]\ \Phi \vdash \int_{a:\mathbb{C}} Q(\overline{a}, a)}\ (\mathsf{end}) \qquad \frac{[\Gamma]\ \left(\int^{a:\mathbb{C}} Q(\overline{a}, a)\right), \Phi \vdash P}{[a : \mathbb{C}, \Gamma]\ Q(\overline{a}, a), \Phi \vdash P}\ (\mathsf{coend})$$

$$\boxed{[\Gamma]\ \Phi \vdash \alpha = \beta : P} \qquad \frac{}{[z : \mathbb{C}, \Gamma]\ k : \Phi(\overline{z}, z) \vdash J(h)[\mathsf{refl}_{\mathbb{C}}] = h : P(\overline{z}, z)}\ (J\text{-}\mathsf{comp})$$

$$\frac{[z : \mathbb{C}, \Gamma]\ \Phi(\overline{z}, z) \vdash \alpha[\mathsf{refl}_{\mathbb{C}}] = \beta[\mathsf{refl}_{\mathbb{C}}] : P(\overline{z}, z)}{[a : \mathbb{C}^{\mathsf{op}}, b : \mathbb{C}, \Gamma]\ e : \hom_{\mathbb{C}}(a, b), \Phi(\overline{b}, \overline{a}) \vdash \alpha[e] = \beta[e] : P(a, b)}\ (J\text{-}\mathsf{eq})$$

$$\frac{\begin{array}{r}[a : \Delta^{\mathsf{op}}, b : \Delta, x : \Gamma] \qquad \Phi(\overline{x}, x, a, b) \vdash \alpha \quad : P(a, b) \\ [z : \Delta, x : \Gamma]\ k : P(\overline{z}, z), \Phi(\overline{x}, x, \overline{z}, z) \vdash \gamma[k] \ : Q(\overline{z}, z) \\ [a : \Delta^{\mathsf{op}}, b : \Delta, x : \Gamma]\ k : Q(a, b), \Phi(\overline{x}, x, \overline{a}, \overline{b}) \vdash \beta[k] : R(a, b)\end{array}}{[z : \Delta, x : \Gamma] \qquad \Phi(\overline{x}, x, \overline{z}, z) \vdash (\beta[\gamma])[\alpha] = \beta[\gamma[\alpha]] : R(\overline{z}, z)}\ (\mathsf{cut\text{-}assoc})$$

Figure 1: Main rules for entailments of first-order dinatural directed type theory.

Implication in the logic is given by the notion of *polarized exponentials* [5, 3], which are interpreted via the pointwise hom of endoprofunctors in Set: their behaviour is captured in the rule (exp), which intuitively allows for predicates to switch between the two sides of the turnstile simply by inverting the variance of all their variables.

(Co)ends as quantifiers. Moreover, we show how dinaturality allows us to more precisely view (co)ends [14] as the "directed quantifiers" of directed type theory, which we present in a correspondence reminiscent of the quantifiers-as-adjoints paradigm of Lawvere [12]. The rules for (co)ends are captured as (coend), (end). Despite the lack of general composition, the rules for directed equality and coends-as-quantifiers can be used to give concise proofs of theorems in category theory using a distinctly *logical* flavour via a series of isomorphisms: e.g., the (co)Yoneda lemma, Kan extensions computed via (co)ends are adjoints, presheaves form a closed category, hom preserves (co)limits, and Fubini, which easily follow by modularly using the logical rules of each connective. This constitutes a concrete step towards formally understanding

the so-called "(co)end calculus" [14] from a logical perspective. We show an example of (co)end calculus as directed type theory via the following two derivations, which formally capture the Yoneda and coYoneda lemma, respectively, using the rules given in Figure 1:

$$\dfrac{\dfrac{\dfrac{[a:\mathbb{C}]\ \Phi(a) \vdash \int_{x:\mathbb{C}} \hom_{\mathbb{C}}^{\mathsf{op}}(a,\overline{x}) \Rightarrow P(x)}{[a:\mathbb{C}, x:\mathbb{C}]\ \Phi(a) \vdash \hom_{\mathbb{C}}^{\mathsf{op}}(a,\overline{x}) \Rightarrow P(x)}\ (\mathsf{end})}{[a:\mathbb{C}]\ \hom_{\mathbb{C}}(\overline{a},x) \times \Phi(a) \vdash P(x)}\ (\mathsf{exp})}{[z:\mathbb{C}]\ \Phi(z) \vdash P(z)}\ (J) \qquad \dfrac{\dfrac{\dfrac{[a:\mathbb{C}]\ \int^{x:\mathbb{C}} \hom_{\mathbb{C}}(\overline{x},a) \times P(x) \vdash \Phi(a)}{[a:\mathbb{C}, x:\mathbb{C}]\ \hom_{\mathbb{C}}(\overline{a},x) \times P(a) \vdash \Phi(x)}\ (\mathsf{coend})}{[z:\mathbb{C}]\ P(z) \vdash \Phi(z)}\ (J)$$

Related works. North [16] describes a dependent directed type theory with semantics in Cat, but using groupoidal structure to deal with the problem of variance in both introduction and elimination rules for directed equality. A similar approach is followed in [2] using the notion of neutral contexts (i.e. groupoidal) instead of core-types. We focus on non-dependent semantics, and tackle the variance issue precisely with the notion of dinatural transformation instead of having to resort back to groupoidal structure.

New and Licata [15] give a sound and complete presentation for the internal language of (hyperdoctrines of) certain virtual equipments. These models capture enriched, internal, and fibered categories, and have an intrinsically directed flavour. This generality comes at the cost of a non-standard syntactic structure of the logic, which forces variables to appear in an ordered linear way. Our work is similar in spirit since we provide a formal setting to prove category-theoretical theorems using logical methods, but we only focus on the 1-category model. We treat ends and coends as quantifiers *directly*, with adjoint-like correspondences to weakening functors acting on the context, without the need for quantifiers to include (restricted forms of) conjunction and implication as in their work. Our rules for directed equality are more reminiscent of the J-rule in MLTT, and specifically focus on the semantic justification based on dinaturality; because of dinaturality, we are similarly restricted in the way that composition can be performed, and indeed the specific substitution structure of Prof, viewed as a double category, is an instance of our cut rules. Since we consider less general models, our contexts do not have any restriction on appearance of variables: this allows us to consider profunctors of many variables as typically needed in (co)end calculus (e.g. Fubini) and to *state* the statement that directed equality can be symmetric. Moreover, certain derivations, e.g., the fact that presheaf categories are cartesian closed, are not easy to capture as abstract properties of such models, while they are straightforward to capture logically using dinaturality.

Another approach to directed type theory involves using geometric models with different flavours [17, 6, 7, 20], typically by axiomatizing a directed interval type. In comparison, we work directly with the more "algebraic" and elementary notions of 1-categories and dinatural transformations, interpreting directed equality directly with hom-functors and their elimination rules rather than with synthetic intervals.

Directed equality can also be captured at the judgemental level [13, 1]; however, such rewrites are typically not described internally using hom-types and their elimination rule.

Future work. We believe type dependency to be the most interesting development of this theory, with particular attention to how the polarity of variables interacts with type dependency. To that end we are currently exploring a notion of *dinatural context extension* based on a "diagonal" generalization of the Grothendieck construction to dinatural families of types $A : \Gamma^{\mathsf{op}} \times \Gamma \to \mathsf{Cat}$, where objects are pairs $(X \in \Gamma, a \in A(X,X))$, and arrows are given by a lax wedge-like condition as pairs $(X,a) \to (Y,b) := (f : X \to Y, \alpha : A(\mathsf{id}_X, f)(a) \to A(f, \mathsf{id}_Y)(b))$.

References

[1] Benedikt Ahrens, Paige Randall North, and Niels van der Weide. Bicategorical type theory: semantics and syntax. *Mathematical Structures in Computer Science*, pages 1–45, October 2023.

[2] Thorsten Altenkirch and Jacob Neumann. Synthetic 1-Categories in Directed Type Theory, October 2024. arXiv:2410.19520.

[3] Edwin S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, January 1990.

[4] Eduardo Dubuc and Ross Street. Dinatural transformations. In S. MacLane, H. Applegate, M. Barr, B. Day, E. Dubuc, Phreilambud, A. Pultr, R. Street, M. Tierney, and S. Swierczkowski, editors, *Reports of the Midwest Category Seminar IV*, Lecture Notes in Mathematics, pages 126–137, Berlin, Heidelberg, 1970. Springer.

[5] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Normal Forms and Cut-Free Proofs as Natural Transformations. In Yiannis N. Moschovakis, editor, *Logic from Computer Science*, pages 217–241, New York, NY, 1992. Springer.

[6] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. Directed univalence in simplicial homotopy type theory, July 2024. arXiv:2407.09146.

[7] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. The Yoneda embedding in simplicial type theory, January 2025. arXiv:2501.13229.

[8] Martin Hofmann. Syntax and Semantics of Dependent Types. In Andrew M. Pitts and Peter Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, Cambridge, 1997.

[9] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In Giovanni Sambin and Jan M Smith, editors, *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, October 1998.

[10] Bart P. F. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1999.

[11] Andrea Laretto, Fosco Loregian, and Niccolò Veltri. Directed equality with dinaturality. September 2024. arXiv:2409.10237.

[12] F. William Lawvere. Adjointness in Foundations. *Dialectica*, 23(3/4):281–296, 1969.

[13] Daniel R. Licata and Robert Harper. 2-Dimensional Directed Type Theory. *Electronic Notes in Theoretical Computer Science*, 276:263–289, September 2011.

[14] Fosco Loregian. *(Co)end Calculus*. London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, 2021.

[15] Max S. New and Daniel R. Licata. A Formal Logic for Formal Category Theory. In Orna Kupferman and Pawel Sobocinski, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 113–134. Springer Nature Switzerland, 2023.

[16] Paige Randall North. Towards a Directed Homotopy Type Theory. *Electronic Notes in Theoretical Computer Science*, 347:223–239, November 2019.

[17] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories. *Higher structures*, 1(1), 2017. arXiv:1705.07442.

[18] Alessio Santamaria. *Towards a Godement calculus for dinatural transformations*. PhD thesis, University of Bath, 2019.

[19] Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, October 2010.

[20] Matthew Z. Weaver and Daniel R. Licata. A Constructive Model of Directed Univalence in Bicubical Sets. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, pages 915–928, New York, NY, USA, July 2020. Association for Computing Machinery.

A Type Theory for Comprehension Categories with Applications to Subtyping

Niyousha Najmaei[1], Niels van der Weide[2], Benedikt Ahrens[3], Paige Randall North[4]

[1] École Polytechnique, Palaiseau, France
[2] Radboud University Nijmegen, The Netherlands
[3] Delft University of Technology, The Netherlands
[4] Utrecht University The Netherlands

We develop a type theory that we show is an internal language for comprehension categories. Usually, the semantics of Martin-Löf type theory (MLTT) is given in discrete or full comprehension categories. Requiring a comprehension category to be full or discrete can be understood as removing one 'dimension' of morphisms. In our syntax, we recover this extra dimension. We show that this extra dimension can be used naturally to encode subtyping as sketched by Coraglia and Emmenegger [5]. We then extend our type theory with Π-, Σ- and Id-types and discuss how to add subtyping for these type formers to both the syntax and semantics.

Motivation There are two primary approaches to studying denotational semantics of a type theory: one starts with syntax and later develops semantics (e.g., simply typed lambda calculus [3], MLTT [9]), while the other begins with intended semantics and then creates a syntax for it (e.g., cubical type theory [2, 4]). One can think of the latter as developing an internal language for a given categorical structure. More specifically, this can be thought of as developing a class of languages that can be used to soundly reason about the given categorical structure. Each instance of the categorical structure then gives a specific language, called its internal language.

We enact the semantics-first process by developing an internal language for comprehension categories. We focus on comprehension categories as they constitute the most general semantics for dependent type theory in the sense that all other categorical structures that are used to interpret dependent theory embed in and can be compared in comprehension categories [1]. However, the structural rules of Martin-Löf dependent type theory (strMLTT) are far from being complete with respect to comprehension categories – indeed, one often restricts to full or discrete comprehension categories to give semantics for strMLTT. Instead of restricting comprehension categories to full or discrete ones, we enlarge the syntax to describe all comprehension categories.

A comprehension category consists, among other things, of two categories: one whose objects interpret the contexts and one which interprets types. Thus, at first glance, a comprehension category can express morphisms between both contexts and types. In strMLTT, however, type morphisms can be recovered from the context morphisms. Both the requirements of discreteness and fullness 'kill off' this 'extra dimension' of morphisms. By not postulating discreteness or fullness, we gain back this 'extra dimension' in the syntax. As argued by Coraglia and Emmenegger [5], morphisms between types can be used to encode subtyping in a natural way. Thus, our syntax and semantics can capture coercive subtyping.

CCTT We design judgements and structural rules, which we call CCTT. The judgements of CCTT include the following two judgements which are not present in strMLTT.

1. $\Gamma \vdash s : \Delta$, where Γ, Δ ctx
2. $\Gamma \mid A \vdash t : B$, where $\Gamma \vdash A, B$ type

Judgement 1 adds explicit substitution to the syntax in the sense of [6]. Judgement 2 expresses 't is a witness for A being a subtype of B'. Judgements 1 and 2 are interpreted as the morphisms in the category of contexts and types in a comprehension category, respectively.

We want types (contexts) and type morphisms (substitutions) to form a category. The rules of CCTT regarding type morphisms are as follows.

$$\frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma \mid A \vdash 1_A : A}\ \textsf{ty-mor-id} \qquad \frac{\Gamma \vdash A, B, C\ \mathsf{type} \quad \Gamma \mid A \vdash t : B \quad \Gamma \mid B \vdash t' : C}{\Gamma \mid A \vdash t' \circ t : C}\ \textsf{ty-mor-comp}$$

$$\frac{\Gamma \mid A \vdash t : B}{\begin{array}{l}\Gamma \mid A \vdash t \circ 1_A \equiv t : B \\ \Gamma \mid A \vdash 1_B \circ t \equiv t : B\end{array}}\ \textsf{ty-id-unit} \qquad \frac{\Gamma \mid A \vdash t : B \quad \Gamma \mid B \vdash t' : C \quad \Gamma \mid C \vdash t'' : D}{\Gamma \mid A \vdash t'' \circ (t' \circ t) \equiv (t'' \circ t') \circ t : D}\ \textsf{ty-comp-assoc}$$

We have similar rules for context morphisms. The rules for context extension and substitution mirror the action of the comprehension functor and the reindexing functors in a comprehension category, respectively. Some of these rules are as follows:

$$\frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma.A\ \mathsf{ctx}}\ \textsf{ext-ty} \qquad \frac{\Gamma \vdash A, B\ \mathsf{type} \quad \Gamma \mid A \vdash t : B}{\Gamma.A \vdash \Gamma.t : \Gamma.B}\ \textsf{ext-tm} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma.A \vdash \pi_A : \Gamma}\ \textsf{ext-proj}$$

$$\frac{\Gamma, \Delta\ \mathsf{ctx} \quad \Gamma \vdash s : \Delta \quad \Delta \vdash A\ \mathsf{type}}{\Gamma \vdash A[s]\ \mathsf{type}}\ \textsf{sub-ty} \qquad \frac{\Delta \vdash A, B\ \mathsf{type} \quad \Gamma \vdash s : \Delta \quad \Delta \mid A \vdash t : B}{\Gamma \mid A[s] \vdash t[s] : B[s]}\ \textsf{sub-tm}$$

In CCTT, the term judgement $\Gamma \vdash a : A$ is not a primitive; instead, we introduce a notation for it that stands for two judgements. This notation mirrors terms being interpreted as sections of the projection morphisms $\pi_A : \Gamma.A \to \Gamma$ in a comprehension category.

Notation 1. $\Gamma \vdash a : A$ stands for the following two judgements:

1. $\Gamma \vdash a : \Gamma.A$
2. $\Gamma \vdash \pi_A \circ a \equiv 1_\Gamma : \Gamma$

Theorem 2 (Soundness)**.** *Every comprehension category models the rules of* CCTT*.*

Subtyping One can regard type morphisms as witnesses of coercive subtyping: a judgement $\Gamma \mid A \vdash t : B$ can be seen as t is a witness for the subtyping relation $A \leq B$. Coraglia and Emmenegger explore this in generalized categories with families (GCwFs), a structure equivalent to comprehension categories [5]. In their notation, this judgement is written as $\Gamma \vdash A \leq_t B$.

Proposition 3. From the rules of CCTT, we can derive the following rule.

$$\frac{\Gamma \vdash A, B\ \mathsf{type} \quad \Gamma \vdash A \leq_t B \quad \Gamma \vdash a : A}{\Gamma \vdash \Gamma.t \circ a : B}$$

The rule in proposition 3 states that if A is a subtype of B, then a term of type A can be coerced to a term of type B. This corresponds to the subsumption rule in coercive subtyping.

In the following table, we discuss the meaning of some of the rules of CCTT from the subtyping perspective, and how they relate to the rules discussed in [5].

Rule of CCTT	Meaning under Subtyping	Rule in [5]
ty-mor-id	Reflexivity of subtyping witnessed by 1_A	-
ty-mor-comp	$A \leq_f B$ and $B \leq_g C$ give $A \leq_{g \circ f} C$.	*Trans* and *Sbsm*
ty-id-unit	Each 1_A is an identity for witness composition.	-
ty-comp-assoc	Composition of witnesses is associative.	-
ext-tm	$A \leq_t B$ gives a context morphism $\Gamma.A \vdash \Gamma.t : \Gamma.B$.	-
sub-tm	Substitution preserves subtyping.	*Wkn* and *Sbst*

Type Formers We extend CCTT with Π-, Σ- and Id-types, and show that these extensions can be interpreted in any comprehension category with suitable structure for each type former. We then discuss how CCTT can be extended with subtyping for each type former, and define a suitable semantic structure for interpreting these extensions.

Related Work Coercive and subsumptive subtyping have been studied from different angles.

Coraglia and Emmenegger [5] observe that vertical morphisms in GCwFs can be seen as witnesses for coercive subtyping. They also show this in the presence of Π- and Σ-types.

Luo and Adams [8] study structural coercive subtyping syntactically. They address coherence issues of composition of subtyping by having functoriality for type formers.

Laurent, Lennon-Bertrand and Maillard [7] extend MLTT to a type theory with functorial type formers. They use this functoriality to extend MLTT to two type theories with coercive and subsumptive subtyping, respectively. They also study meta-theoretic properties of their systems, e.g. showing that their functorial system is normalizing and has decidable type checking.

Zeilberger and Melliès [10] give a categorical view of subsumptive subtyping. They interpret type systems as functors from a category of type derivations to a category of underlying terms. In this setting, subtyping derivations are vertical morphisms.

References

[1] Benedikt Ahrens, Peter LeFanu Lumsdaine, and Paige Randall North. Comparing semantic frameworks for dependently-sorted algebraic theories. In Oleg Kiselyov, editor, *Programming Languages and Systems - 22nd Asian Symposium, APLAS 2024, Kyoto, Japan, October 22-24, 2024, Proceedings*, volume 15194 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2024.

[2] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, volume 26 of *LIPIcs*, pages 107–128. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[3] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[4] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.

[5] Greta Coraglia and Jacopo Emmenegger. Categorical Models of Subtyping. In Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg, editors, *29th International Conference on Types for Proofs and Programs (TYPES 2023)*, volume 303 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] Pierre-Louis Curien, Richard Garner, and Martin Hofmann. Revisiting the categorical interpretation of dependent type theory. *Theor. Comput. Sci.*, 546:99–119, 2014.

[7] Théo Laurent, Meven Lennon-Bertrand, and Kenji Maillard. Definitional functoriality for dependent (sub)types. volume 14576 of *Lecture Notes in Computer Science*, pages 302–331. Springer, 2024.

[8] Zhaohui Luo and Robin Adams. Structural subtyping for inductive types with functorial equality rules. *Math. Struct. Comput. Sci.*, 18(5):931–972, 2008.

[9] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

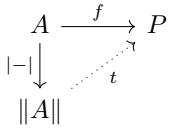
[10] Paul-André Melliès and Noam Zeilberger. Functors are type refinement systems. pages 3–16. ACM, 2015.

Irregular models of type theory

Andrew W Swan

University of Ljubljana
wakelin.swan@gmail.com

Propositional truncation is one of the most basic examples of higher inductive types, and formalises the idea of a proposition "most closely approximating a type" [Uni13]. We can define the propositional truncation of a type A as follows. It is a type $\|A\|$ together with a map $|-| : A \to \|A\|$, such that $\|A\|$ is a proposition and satisfies the following recursion principle: given any proposition P and map $f : A \to P$, there is a (unique) map $t : \|A\| \to P$, as illustrated below.



We will give two general techniques for constructing models of type theory that do *not* have propositional truncation, while retaining as much of the other parts of type theory as possible. Both techniques are based on higher modalities [RSS20], and in particular to get concrete examples of models, we will use higher modalities on cubical assemblies [Uem19, SU21] arising from Lifschitz realizability [vO08, LvO13, RS20, Kou19]. To explain the key property of Lifschitz realizability that we will use, we first recall that for a fixed universe $\mathcal{U}$, we can think of a modality as a Σ-*closed reflective subuniverse* of $\mathcal{U}$, denoted $\mathcal{U}_\bigcirc \hookrightarrow \mathcal{U}$ [RSS20, Section 1.3]. We recall [Qui16] that lex Σ-closed reflective subuniverses can be viewed as models of homotopy type theory. We note however, that the implementation of higher inductive types, and in particular propositional truncation, may or may not be preserved by the inclusion $\mathcal{U}_\bigcirc \hookrightarrow \mathcal{U}$. In particular for modalities arising from Lifschitz realizability, we can show the following theorem.

Theorem 1. *Let $\bigcirc$ be the Lifschitz realizability modality on cubical assemblies. Then,*

1. *The inclusion $\mathcal{U}_\bigcirc \hookrightarrow \mathcal{U}$ preserves Π and Σ types.*
2. *The inclusion $\mathcal{U}_\bigcirc \hookrightarrow \mathcal{U}$ preserves the empty type, coproducts and all W-types.*
3. *The inclusion $\mathcal{U}_\bigcirc \hookrightarrow \mathcal{U}$ does* not *preserve propositional truncation.*

Moreover, following [LvO13, RS20] we can construct a countable increasing sequence of reflective subuniverses $\bigcirc_n$ for $n \geq 2$ as variations on Lifschitz realizability. One can then show the following theorem.

Theorem 2. *For each $n \geq 2$,*

The inclusion $\mathcal{U}_{\bigcirc_n} \hookrightarrow \mathcal{U}_{\bigcirc_{n+1}}$ preserves Π and Σ types.

The inclusion $\mathcal{U}_{\bigcirc_n} \hookrightarrow \mathcal{U}_{\bigcirc_{n+1}}$ preserves the empty type, coproducts and all W-types.

The inclusion $\mathcal{U}_{\bigcirc_n} \hookrightarrow \mathcal{U}_{\bigcirc_{n+1}}$ does not *preserve propositional truncation.*

This already suffices to construct our simplest model of type theory without propositional truncation. We recall that when viewing locally cartesian closed categories as models of extensional type theory, the existence of propositional truncation in the type theory corresponds precisely to the locally cartesian closed category being regular [AB04, Mai05]. By simply taking the colimit over the increasing sequence of reflective subuniverses in theorem 2, one can obtain the following.

Theorem 3. *There is a category $\mathcal{E}$ such that*

1. *$\mathcal{E}$ is locally cartesian closed.*
2. *$\mathcal{E}$ has an initial object, disjoint coproducts and all W-types (including in particular a natural number object).*
3. *$\mathcal{E}$ is not regular.*

The simpler construction is limited in that it does not feature any universes of small types. Adding universes introduces multiple complications. Firstly, instead of producing models of extensional type theory, we might alternatively ask for the universe to be univalent, to get models of univalent type theory. Secondly, we might ask to have not just one universe, but a cumulative increasing sequence of universes. Finally, working with universes introduces the following subtle point about the definition of propositional truncation. For a given type $A : \mathcal{U}_n$, it might happen that A has a *restricted* propositional truncation in the sense that it does not satisfy the recursion principle in general (i.e. for all universe levels) but does once we restrict to propositions at universe level n. That is, for any proposition $P : \mathcal{U}_n$, and map $A \to P$, there is a map $\|A\| \to P$. To get models avoiding even this weaker version of propositional truncation, we use our second, more sophisticated construction, which combines the previous ideas with gluing [Shu14].

In particular, we have the following key lemma:

Lemma 4. *Suppose that $\mathcal{E}$ is a type theoretic fibration category in the sense of [Shu14] and that it moreover contains an inclusion of subuniverses $\mathcal{U} \hookrightarrow \mathcal{V}$ which preserves all type constructors except for propositional truncation, but does not preserve propositional truncation. Then viewing $\mathcal{E}^{\to}$ as a type theoretic fibration category, again as in [Shu14], it contains a universe closed under all type constructors except propositional truncation, and does not satisfy restricted propositional truncation.*

Using this and the results above on Lifschitz realizability, we can obtain the following theorems:

Theorem 5. *There is a model of type theory with a cumulative increasing sequence of universes such that:*

1. *Each universe is univalent.*
2. *Each universe is closed under Π and Σ types, coproducts and W-types, and contains the empty type.*
3. *None of the universes have all restricted propositional truncations.*

Theorem 6. *There is a model of extensional type theory with a cumulative increasing sequence of universes such that:*

1. *Each universe is closed under Π and Σ types, coproducts and W-types, and contains the empty type.*
2. *None of the universes have all restricted propositional truncations.*

As well as talking about these results, I will report on the current work in progress of adding certain higher inductive types, such as the circle type to the model.

References

[AB04] Steven Awodey and Andrej Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

[Kou19] Dimitrios Koutsoulis. Lifschitz realizability for homotopy type theory. Master's thesis, Institute for Logic, Language and Computation, University of Amsterdam, 2019. Available at https://eprints.illc.uva.nl/1756/1/MoL-2019-26.text.pdf.

[LvO13] Sori Lee and Jaap van Oosten. Basic subtoposes of the effective topos. *Annals of Pure and Applied Logic*, 164(9):866 – 883, 2013.

[Mai05] Maria Emilia Maietti. Modular correspondence between dependent type theories and categories including pretopoi and topoi. *Mathematical Structures in Computer Science*, 15:1089–1149, 12 2005.

[Qui16] Kevin Quirin. *Lawvere-Tierney sheafification in Homotopy Type Theory*. Theses, Ecole des Mines de Nantes, December 2016.

[RS20] Michael Rathjen and Andrew W. Swan. Lifschitz realizability as a topological construction. *The Journal of Symbolic Logic*, 85(4):1342–1375, 2020.

[RSS20] Egbert Rijke, Michael Shulman, and Bas Spitters. Modalities in homotopy type theory. *Logical Methods in Computer Science*, Volume 16, Issue 1, January 2020.

[Shu14] Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, 25(5):1–75, November 2014.

[SU21] Andrew W. Swan and Taichi Uemura. On Church's thesis in cubical assemblies. *Mathematical Structures in Computer Science*, 31(10):1185–1204, 2021.

[Uem19] Taichi Uemura. Cubical Assemblies, a Univalent and Impredicative Universe and a Failure of Propositional Resizing. In Peter Dybjer, José Espírito Santo, and Luís Pinto, editors, *24th International Conference on Types for Proofs and Programs (TYPES 2018)*, volume 130 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:20, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Uni13] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[vO08] Jaap van Oosten. *Realizability: An Introduction to its Categorical Side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, North Holland, 2008.

An algebraic internal groupoidal model of Martin-Löf type theory

Calum Hughes

University of Manchester
calum.hughes@postgrad.manchester.ac.uk

Abstract

The groupoidal model of Martin-Löf type theory was introduced by Hofmann and Streicher in 1998, proving that the uniqueness of identity proofs property is false in this type theory and thus revealing higher dimensional structure [2]. In this model, dependent types are modelled by isofibrations between groupoids; these are functors with the *property* that paths can be lifted in a suitable way. Groupoids and isofibrations form a category with families, which gives a model of dependent type theory.

One issue with formalising models given by categories with families constructively is that substitution of terms is only modelled up to isomorphism rather than on the nose. Due to work by Gambino and Larrea [1], this problem can be ironed out by working with structure rather than mere properties.

In this talk, I will describe some recent work in which this algebraic approach is taken to construct a structural version of Hofmann and Streicher's groupoidal model. We work with cloven isofibrations; these are functors equipped with the *structure* of a path lifting. Moreover, we work in the setting of internal groupoid theory, without the reliance on set theoretic foundations. This recovers Hofmann and Streicher's approach when working internally to the category of sets and forgetting the algebraic structure. Our abstract setting allows for a relative consistency result: that models of Martin-Löf type theory can be constructed using Martin-Löf type theory.

References

[1] Nicola Gambino and Marco Federico Larrea. Models of Martin-Löf type theory from algebraic weak factorisation systems, *The Journal of Symbolic Logic*, 88(1):242–289, 2023.

[2] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. *Twenty- five years of constructive type theory (Venice, 1995)*, 36:83–111, 1998.

Realizability Triposes from Sheaves

Bruno da Rocha Paiva and Vincent Rahli

University of Birmingham, United Kingdom

Abstract

Given a topos $\mathcal{E}$ and a Lawvere-Tierney topology $\square : \Omega \to \Omega$ on it, we develop a realizability $\mathcal{E}$-tripos using the internal logic of the topos. Instantiating $\mathcal{E}$ with a category of presheaves, we recover a notion of realizability with choice sequences.

Choice sequences first appeared in *Brouwer's second act of intuitionism* [14]. Brouwer envisioned an idealised mathematician that would generate entries of an infinitely proceeding sequence $(\alpha_0, \alpha_1, \alpha_2, \dots)$. At any given moment, the mathematician would only have access to the entries generated so far, hence any deductions would necessarily rely on a finite number of entries. The first formal systems of Brouwer's intuitionism were developed by Kleene and Vesley [9] and Kreisel and Troelstra [10] in which the authors investigated the Bar Theorem, continuity principles, as well as different kinds of choice sequences.

More recently, interpretations of Brouwer's choice sequences have been leveraged to give anti-classical models of dependent type theories. In [4] the authors give a computational account of forcing. Based on this view of forcing the authors of [5] produce a model of MLTT falsifying Markov's principle. This interpretation of choice sequences is combined with term models in a series of papers [1, 2, 3, 11, 7] to explore principles such as bar induction, continuity of functions on the Baire space and different versions of Markov's principle. As pointed out by [12], there is a common thread of constructions internal to sheaf models which links all the foregoing works. In the tradition of Kripke and Beth semantics, by taking a category of sheaves over a preordered set $\mathbb{W}$ of worlds and carrying out the standard operational constructions internally to this model, we should expect to recover models akin to the above.

In what follows, we will start from this observation and attempt to connect these realizability constructions, in the form of PER models with choice sequences, to categorical realizability over categories of sheaves, rather than the category of sets. As in the abstract, we fix a topos $\mathcal{E}$ and a Lawvere-Tierney topology $\square : \Omega \to \Omega$ and proceed to define a realizability tripos over $\mathcal{E}$. We refer to [8] for an introduction to topos theory, in particular section A4.4 for a treatment of Lawvere-Tierney topologies.

Definition 1. *Given objects X and Y of $\mathcal{E}$, we define **partial morphisms from X to Y** as morphisms from X to $Y_\perp$, where $Y_\perp$ is the partial map classifier of Y [8, §A2.4]. In the internal logic, given elements x and y of $X_\perp$, we use $x\downarrow$ to mean that x is defined, $x \preccurlyeq y$ to mean that if x is defined then so is y and their values agree, and $x \simeq y$ for the conjunction of $x \preccurlyeq y$ and $y \preccurlyeq x$.*

Definition 2. *An **internal partial combinatory algebra** consists of an object A of $\mathcal{E}$, a partial morphism $- \cdot - : \mathsf{A} \times \mathsf{A} \rightharpoonup \mathsf{A}$ and elements $\mathsf{k}, \mathsf{s} : \mathsf{A}$ satisfying the internal statements:*

$$\begin{array}{ccc} \mathsf{k} \cdot a \downarrow & \mathsf{s} \cdot a \downarrow & \mathsf{s} \cdot a \cdot b \downarrow \\ a \preccurlyeq \mathsf{k} \cdot a \cdot b & a \cdot c \cdot (b \cdot c) \preccurlyeq \mathsf{s} \cdot a \cdot b \cdot c & \end{array}$$

We define partial combinatory algebras (pca) using $\preccurlyeq$ as opposed to $\simeq$. It is shown in [6] that any "weak" pca, that is using $\preccurlyeq$, is isomorphic to a "strong" pca, that is using $\simeq$, hence this is mainly an aesthetic decision. Note that weak pcas differ from ordered pcas [17, §1.8].

Whereas an ordered pca comes equipped with an ordering on the underlying object A, weak pcas simply use the ordering $\preccurlyeq$ on partial terms $\mathsf{A}_\perp$.

The usual story with partial combinatory algebras carries over to the internal setting. We can still show that a pca is functionally complete and with that we get access to most programming constructs needed for realizability such as pairings, booleans, coproducts, and whatever else the mind might dream of. See [17, §1.1] for an elaboration on pcas and how to program with them.

With an internal pca we could now define a realizability tripos akin to the usual set-based realizability triposes. In fact this is done in [16], in which the author takes a pca internal to a category of presheaves, takes its sheafification to get a pca internal to the relevant category of sheaves, and then implicitly works in the realizability topos arising out of said pca object.

Definition 3. *Given an object X we define the type of* ***realizability predicates on*** *X as the type $X \to \mathcal{P}\mathsf{A}$. We further define an ordering on realizability predicates $\varphi, \psi : X \to \mathcal{P}\mathsf{A}$ by*

$$\varphi \leq \psi :\equiv \exists e : \mathsf{A}.\forall x : X.\forall a \in \varphi(x).\Box(e \cdot a \downarrow \wedge\, e \cdot a \in \psi(x))$$

Given a realizability predicate $\varphi : X \to \mathcal{P}\mathsf{A}$, for each $x : X$ we think of the subobject $\varphi(x) \hookrightarrow \mathsf{A}$ as the programs which evidence that x satisfies φ, i.e. its *realizers*. We say that φ implies ψ, written $\varphi \leq \psi$, if there exists a program e which for every x, will take evidence that x satisfies φ and convert it to evidence that x satisfies ψ. We include the modality $\Box$ to accommodate for choice sequences. For example suppose the program e relies on the sixth entry of a choice sequence α, but so far we have only generated the first three entries. In such a case, the $\Box$ lets us generate more entries for α before requiring that $e \cdot a$ be defined. If we did not use the modality then we would not be able to generate more entries in α, effectively disallowing the use of choice sequences as realizers.

Using the internal language of $\mathcal{E}$ we can show that realizability predicates on an object X with this particular ordering form a pre-ordered set. Furthermore, we can define reindexing pre-Heyting algebra morphisms by precomposition, show these have left and right adjoints (as monotone maps) satisfying the Beck-Chevalley condition, and finally we can give an appropriate generic element giving us an $\mathcal{E}$-tripos. For an introduction to the theory of realizability triposes and constructions on these we refer the reader to [17, §2]. The definitions of the required left and right adjoints, and generic element are very similar with the ones in the usual setting.

In [15] the authors also use the internal logic of a category $\mathcal{E}$ to define categories of assemblies and show that these still give models of constructive set theories. While the authors assume less about the category $\mathcal{E}$ to define assemblies, they stick to defining assemblies over the internal version of Kleene's first algebra $\mathcal{K}_1$. So while the tools used are similar, the classes of models considered are different.

To talk about choice sequences we instantiate $\mathcal{E}$ to the category of presheaves over a poset $\mathbb{W}$. In the prototypical case where we want our pca to have a single choice sequence code δ, we may take this poset to be the set of lists of natural numbers inversely ordered by prefix. The generated values of δ at each world would then be decided by the underlying list of natural numbers. For the modality $\Box$, we take the Lawvere-Tierney topology associated with the following notion of covering: an upwards closed set $\mathcal{U} \subseteq \mathbb{W}$ covers a world $w : \mathbb{W}$ if all increasing sequences of worlds starting at w intersect with $\mathcal{U}$. The category of assemblies arising out of this tripos seems like a particularly good setting for studying the realizability of choice sequences themselves, it should contain interpretations of the natural numbers and the Baire space while being simpler to work with than the realizability topos.

In this setting, an internal pca will consists of a pca A_w for each world $w : \mathbb{W}$ with application maps $(-) \cdot_w (-)$ and transition maps $(-)|_{w \sqsubseteq v} : \mathsf{A}_v \to \mathsf{A}_w$. The application maps, which are

partial functions, have to be monotone in their domain as well as natural. By choosing different pca objects we expect to be able to handle different versions of choice sequences, from fully lawless choice sequences, to lawful choice sequences and any variation sitting in between these. This contrasts the situation with a sheafified pca object [16], where sheafification inadvertently adds realizers which may not have been computable before. For example, consider the pca object given by $\mathcal{K}_1$ at every world. In particular, application of terms is independent of the world and so we have no choice sequence realizers, as their behaviour will be the same at every world. If we take its sheafification, a realizer at a world w won't be a single element $e : \mathcal{K}_1$ anymore, but will instead be a compatible family of realizers $(e_u)_{u\in\mathcal{U}}$ indexed by a cover of w. Aside from being compatible [8, Definition 2.1.2 in §C2.1], we have no restrictions on this family of realizers, so sheafification has added realizers whose behaviour can vary wildly between different worlds, despite starting with a pca object without choice sequences.

Towards the study of realizability of choice sequences, we now suggest two assemblies of interest: that of pure natural numbers and that of effectful natural numbers.

Definition 4. *The* ***pure natural numbers assembly****, denoted* $\mathsf{N}_{\mathsf{pur}}$*, consists of the constant presheaf* $\Delta\mathbb{N}$ *along with the realizability relation* $e \Vdash_w n$ *if and only if the code* $e : \mathsf{A}_w$ *equals the Church encoding of* $n : \mathbb{N}$ *at* w*.*

Definition 5. *The* ***effectful natural numbers assembly****, denoted* $\mathsf{N}_{\mathsf{eff}}$*, consists of the sheafification of* $\Delta\mathbb{N}$*. As for the realizability relation, if we have* $\mathcal{U}$ *covering* w *and and a compatible family* n_u *of natural numbers, then* $e \Vdash_w (n_u)_{u\in\mathcal{U}}$*, if and only if there exists a cover* $\mathcal{V}$ *of* w *and for all* $v \in \mathcal{U} \cap \mathcal{V}$ *and* $a : \mathsf{A}_v$*,* $(e|_{v\sqsubseteq w}) \cdot_v a$ *is defined and equals the Church encoding of* n_v *at* v*.*

At the level of underlying presheaves, the later is the sheafification of the former, so we hope to find an analogous universal property to justify it as the correct definition of effectful natural numbers. With this, we can then study choice sequences as the exponential object $\mathsf{N}_{\mathsf{eff}} \to \mathsf{N}_{\mathsf{eff}}$ in this category.

We intend on using a theorem prover to formalise the arguments in the internal logic of $\mathcal{E}$ as done in [13]. The formalisation is currently in its early stages, but we hope to progress more on it once definitions are more settled.

References

[1] Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. Computability Beyond Church-Turing via Choice Sequences. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 245–254, New York, NY, USA, July 2018. Association for Computing Machinery.

[2] Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. Open Bar - a Brouwerian Intuitionistic Logic with a Pinch of Excluded Middle. In *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*, volume 183 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:23, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[3] Liron Cohen, Yannick Forster, Dominik Kirst, Bruno Da Rocha Paiva, and Vincent Rahli. Separating Markov's Principles. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–14, Tallinn Estonia, July 2024. Association for Computing Machinery.

[4] Thierry Coquand and Guilhem Jaber. A Computational Interpretation of Forcing in Type Theory. In *Epistemology versus Ontology*, pages 203–213. Springer Netherlands, Dordrecht, 2012.

[5] Thierry Coquand and Bassel Mannaa. The Independence of Markov's Principle in Type Theory. *LIPIcs, Volume 52, FSCD 2016*, 52:17:1–17:18, 2016.

[6] Eric Faber and Jaap Van Oosten. Effective operations of type 2 in PCAs. *Computability*, 5(2):127–146, May 2016.

[7] Yannick Forster, Dominik Kirst, Bruno da Rocha Paiva, and Vincent Rahli. Markov's Principles in Constructive Type Theory. In *29th International Conference on Types for Proofs and Programs*, Valencia, Spain.

[8] Peter T Johnstone. *Sketches of an Elephant A Topos Theory Compendium*. Oxford University Press, Oxford, September 2002.

[9] Stephen Cole Kleene and Richard Eugene Vesley. *The Foundations of Intuitionistic Mathematics: Especially in Relation to Recursive Functions*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1965.

[10] G. Kreisel and A.S. Troelstra. Formal systems for some branches of intuitionistic analysis. *Annals of Mathematical Logic*, 1(3):229–387, 1970.

[11] Vincent Rahli and Mark Bickford. Validating Brouwer's continuity principle for numbers using named exceptions. *Mathematical Structures in Computer Science*, 28(6):942–990, June 2018.

[12] Jonathan Sterling. Higher order functions and Brouwer's thesis. *Journal of Functional Programming*, 31:30, 2021.

[13] Jonathan Sterling and Robert Harper. Guarded Computational Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 879–888, Oxford United Kingdom, July 2018. ACM.

[14] Mark van Atten. *On Brouwer*. Wadsworth Publishing Company, 2004.

[15] Benno Van Den Berg and Ieke Moerdijk. Aspects of predicative algebraic set theory, II: Realizability. *Theoretical Computer Science*, 412(20):1916–1940, April 2011.

[16] Jaap van Oosten. A semantical proof of De Jongh's theorem. *Archive for Mathematical Logic*, 31(2):105–114, March 1991.

[17] Jaap van Oosten. *Realizability: An Introduction to Its Categorical Side*. Number 152 in Studies in Logic and the Foundations of Mathematics. Elsevier, Oxford, 1st ed edition, 2008.

Solving Guarded Domain Equations in Presheaves Over Ordinals and Mechanizing It

Sergei Stepanenko and Amin Timany

Aarhus University, Aarhus, Denmark
{sergei.stepanenko, timany}@cs.au.dk

1 Introduction

Finding solutions to so-called recursive domain equations [11] is a well-known, important problem in the study of programs and programming languages. Mathematically speaking, the problem is finding a fixed point (up to isomorphism) of a suitable endo-functor $F : \mathcal{C} \to \mathcal{C}$ on a suitable category $\mathcal{C}$, *i.e.*, an object X of $\mathcal{C}$ such that $F(X) \simeq X$ [9, 8, 11, 14, 1, 4]. A particularly useful instance, inspired by the step-indexing technique, is where the functor is over (a subcategory of) the category of presheaves over the ordinal ω and the functors are locally-contractive, also known as *guarded functors* [3]. This corresponds to step-indexing over natural numbers. However, for certain problems, *e.g.*, when dealing with infinite non-determinism, one needs to employ trans-finite step-indexing, *i.e.*, consider presheaf categories over higher ordinals [2, 5]. Prior work on trans-finite step-indexing either only considers a very narrow class of functors over a particularly restricted subcategory of presheaves over higher ordinals [12], or treats the problem very generally working with sheaves over an arbitrary complete Heyting algebra with a well-founded basis [3]. In this work we present a solution to the guarded domain equations problem over *all* so-called guarded functors over the category of *presheaves* over ordinal numbers, as well as its mechanization in the Rocq prover. This can be seen as a simplification of the work of Birkedal *et al.* [3] from the setting of the category of sheaves to the setting of the category of presheaves which is more amenable to mechanization using proof assistants. Our Rocq mechanization [13] can be found at: https://github.com/logsem/synthetic_domains.

2 Presheaves Over All Ordinals

In this work we talk about step-indexing over (all) ordinals, *e.g.*, we speak of sheaves or presheaves over **Ord**, the set of all ordinals (which we also consider to be a preorder category under the usual order). This is to be understood as the set of all ordinals definable in a certain Grothendieck universe. (In Rocq, the type **Ord** is a universe polymorphic definition corresponding to the type of all ordinals in the universe.)

An important endo-functor on $\mathbf{PSh}(\mathbf{Ord})$ that plays an important role in our development is the so-called later functor ($\blacktriangleright : \mathbf{PSh}(\mathbf{Ord}) \to \mathbf{PSh}(\mathbf{Ord})$):

$$\blacktriangleright F(\alpha) := \lim_{\beta \prec \alpha} F(\beta) \qquad\qquad (\blacktriangleright F)_{\beta \preceq \alpha} := \lim_{\gamma \prec \beta} \Pi_\gamma^{\blacktriangleright F(\alpha)}$$

The object map of $\blacktriangleright$, at each stage, takes the limit (in **Set**) of the diagram induced by the object (presheaf) it is mapping at all smaller stages. In particular, $\blacktriangleright F(0)$ is always the terminal (singleton) set, and $\blacktriangleright F(\alpha^+) \simeq F(\alpha)$. The morphism map of the functor $\blacktriangleright$, $(\blacktriangleright F)_{\beta \preceq \alpha}$ is defined as the amalgamation of projections $\Pi_\gamma^{\blacktriangleright F(\alpha)} : \blacktriangleright F(\alpha) \to F(\gamma)$ of the limit that is $(\blacktriangleright F)(\alpha)$. There is also an important natural transformation $\mathsf{Next} : \mathsf{id}_{\mathbf{PSh}(\mathbf{Ord})} \to \blacktriangleright$ associated with $\blacktriangleright$.

We say a morphism in $\mathbf{PSh}(\mathbf{Ord})$, *i.e.*, a natural transformation $\eta : F \to G$, is contractive, if it factors through Next, *i.e.*, $\eta = \eta' \circ \mathsf{Next}_F$; we call η' the witness of contractivity. Contractive morphisms are closed under composition with any other (not necessarily contractive) natural transformation. Importantly, contractive natural transformations have *unique* fixed points; a natural transformation $\xi : 1 \to A$ is a fixed point of a contractive morphism $\eta : A \to A$ if $\eta \circ \xi = \xi$.

3 Locally Contractive Functors and Solutions

Definition (Locally Contractive Functors)**.** *Let $\mathcal{C}$ and $\mathcal{D}$ be two $\mathbf{PSh}(\mathbf{Ord})$-enriched categories. We write $\mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{A,B}$ for the object of $\mathbf{PSh}(\mathbf{Ord})$ representing the collection of morphisms from A to B in a category $\mathcal{C}$, and we write $\mathbb{E}^{\mathsf{hm}_F}_{A,B} : \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{A,B} \to \mathbb{E}^{\mathsf{hom}_{\mathcal{D}}}_{F(A),F(B)}$ for the morphism of $\mathbf{PSh}(\mathbf{Ord})$ representing the internal action of an enriched functor F on morphisms. We say a $\mathbf{PSh}(\mathbf{Ord})$-enriched functor $F : \mathcal{C} \to \mathcal{D}$ is locally contractive if the internal action morphisms of F, $\mathbb{E}^{\mathsf{hm}_F}_{A,B}$, are contractive with the witness of contractivity being morphisms $\mathbb{E}^{\blacktriangleright \mathsf{hm}_F}_{A,B} : \blacktriangleright \mathbb{E}^{\mathsf{hom}_{\mathcal{C}}}_{A,B} \to \mathbb{E}^{\mathsf{hom}_{\mathcal{D}}}_{F(A),F(B)}$. Furthermore, expressed in terms of equality of morphisms in $\mathbf{PSh}(\mathbf{Ord})$, the morphisms $\mathbb{E}^{\blacktriangleright \mathsf{hm}_F}_{A,B}$ must preserve identity and composition.*

Importantly, the functor $\blacktriangleright$ itself (under self-enrichment of $\mathbf{PSh}(\mathbf{Ord})$) is locally contractive, and locally contractive functors are closed under composition with arbitrary enriched functors.

Theorem. *Any locally contractive functor $F : \mathcal{C} \to \mathcal{C}$ has a unique (up to isomorphism) solution X for which $F(X) \simeq X$.*

Note how each solution $F(X) \simeq X$ gives rise to an F-algebra structure on X. The proof of uniqueness of the solution essentially shows that an object is a solution if and only if it is the initial F-algebra — the F-algebra morphism out of this initial F-algebra is constructed as the unique fixed point of the internal action of the functor F on morphisms, which is contractive since F is locally contractive. It is also for this reason that our construction of the solution is essentially constructing an F-algebra whose underlying map is an isomorphism. Technically, this construction consists of iteratively (by transfinite induction) constructing ordinal-shaped diagrams of F-algebras, taking their limit, and applying F to the constructed limit.

Apart from working with the category of F-algebras as opposed to working directly in $\mathcal{C}$, our solution construction differs from that of Birkedal *et al.* [3] in how we treat zero and limit ordinals. Working with sheaves, Birkedal *et al.* [3] at zero and limit ordinals simply take the limit of the construction at stages below. By contrast, we apply F to the limit at every single stage and not just at successor ordinals. (A similar difference also appears in our construction of fixed points of contractive morphisms as defined above.) Another way to look at this difference is if we look at the sequence of objects constructed in these two approaches (in our case the carrier objects of the algebras we compute). Up to isomorphism, what we compute is the sequence X while Birkedal *et al.* [3] compute the sequence Y:

$$X_0 := F(1); \quad X_1 := F(F(1)); \quad X_2 := F(F(F(1))); \quad \cdots \quad X_\omega := F(\lim_{\alpha \prec \omega} X_\alpha); \quad X_{\omega+} := F(X_\omega); \quad \cdots$$

$$Y_0 := 1; \quad Y_1 := F(1); \quad Y_2 := F(F(1)); \quad \cdots \quad Y_\omega := \lim_{\alpha \prec \omega} Y_\alpha; \quad Y_{\omega+} := F(Y_\omega); \quad \cdots$$

4 Related Work

The most closely related works to us are Rocq mechanizations of the domain equation solver of the ModuRes library [10], the domain equation solver of the Iris program logic [6] which

is a nicer reimplementation of the domain equation solver of the ModuRes library, and the domain equation solver of transfinite Iris [12]. (In fact, for our mechanization we have used the step-indexing development of Spies *et al.* [12] who use the mechanization of ordinal numbers by Kirst *et al.* [7].) The former two mechanizations work with the category of complete ordered family of equivalences (COFEs), a representation of the category of complete bisected bounded ultra metric spaces (CBUlt) [4] that is particularly amenable to mechanizations [10]. These only support step-indexing up to ω. Transfinite Iris, inspired by Birkedal *et al.* [3], extends the definition of OFEs (COFEs without completeness requirement) and COFEs to higher ordinals. However, Transfinite Iris, unlike the ModuRes library and Iris, only solves domain equations for functors of the form $\mathrm{OFE}^{\mathsf{op}} \times \mathrm{OFE} \to \mathrm{COFE}$ and not $\mathrm{COFE}^{\mathsf{op}} \times \mathrm{COFE} \to \mathrm{COFE}$.

References

[1] Pierre America and Jan J. M. M. Rutten. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *J. Comput. Syst. Sci.*, 39(3):343–375, 1989.

[2] Lars Birkedal, Ales Bizjak, and Jan Schwinghammer. Step-Indexed Relational Reasoning for Countable Nondeterminism. *Log. Methods Comput. Sci.*, 9(4), 2013.

[3] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 55–64. IEEE Computer Society, 2011.

[4] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theor. Comput. Sci.*, 411(47):4102–4122, 2010.

[5] Ales Bizjak, Lars Birkedal, and Marino Miculan. A Model of Countable Nondeterminism in Guarded Type Theory. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2014.

[6] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM, 2016.

[7] Dominik Kirst and Gert Smolka. Large model constructions for second-order ZF in dependent type theory. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 228–239. ACM, 2018.

[8] Dana Scott. Outline of a mathematical theory of computation. Technical Report PRG02, OUCL, November 1970.

[9] Dana Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136, Berlin, Heidelberg, 1972. Springer Berlin Heidelberg.

[10] Filip Sieczkowski, Ales Bizjak, and Lars Birkedal. ModuRes: A Coq Library for Modular Reasoning About Concurrent Higher-Order Imperative Programming Languages. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2015.

[11] Michael B. Smyth and Gordon D. Plotkin. The Category-Theoretic Solution of Recursive Domain Equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

[12] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite Iris: resolving an existential dilemma of step-indexed separation

logic. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 80–95. ACM, 2021.

[13] Sergei Stepanenko and Amin Timany. The Rocq Mechanization of Solving Guarded Domain Equations in Presheaves Over Ordinals. https://github.com/logsem/synthetic_domains.

[14] Mitchell Wand. Fixed-Point Constructions in Order-Enriched Categories. *Theor. Comput. Sci.*, 8:13–30, 1979.

Algebraic Universes and Variances For All

Matthieu Sozeau[1] and Marc Bezem[2]

[1] LS2N & Inria de l'Université de Rennes,
Nantes, France
matthieu.sozeau@inria.fr
[2] University of Bergen
Bergen, Norway
Marc.Bezem@uib.no

Building on a refinement of the novel loop-checking algorithm of Bezem and Coquand [2], we present a new design and implementation of cumulative universe polymorphism for dependent type theories, materialized in a branch [6] of the Rocq prover. Our system handles universe level polymorphism for the full universe algebra $(\ell, 0, +1, \max)$. The constraint solving algorithm based on Horn clauses is theoretically polynomial. However, to stay competitive with the previous algorithm based on the work of Bender et al. [1], we make the algorithm incremental and integrate a union-find datastructure to capture semantic universe level equalities by efficiently represented equivalence classes; this makes the algorithm more user-friendly in practice, reflecting inferred equalities instantaneously in annotations.

On the type theory side, the original system of Sozeau and Tabareau [9] is simply generalized: universe-polymorphic binders can take universe expressions as arguments, all constraints can be enforced and checked, and we remove a long-standing distinction between inferred types and checkable types [4]. This puts the system on par with Agda and Lean's (non-cumulative) universe systems.

On the elaboration side, the previous system [9] introduced a heuristic in unification to avoid unnecessary unfoldings and an (infamous) minimization procedure to try to reduce the number of unnecessarily quantified universes and constraints, which naturally grows exponentially fast in presence of implicit cumulativity [3]. We generalize the notion of universe variance introduced for cumulative inductive types [10] to all definitions and use it to refine the unification heuristics and provide a more principled universe level metavariable solving process: it should now guarantee a principal typing property.

The talk will present all these aspects along with examples and our preliminary performance benchmarks and compatibility status.

A new loop-checking algorithm for universes

In dependent type theories with predicative universe levels, universe checking involves verifying equalities (inequalities $u \leq v$ are interpreted as $v = \max(u, v)$) between universe level expressions:

Universe levels u, v	$:=$	ℓ	(universe level variable)
	$\mid$	0	(bottom universe)
	$\mid$	$u + 1$	(successor)
	$\mid$	$\max(u, v)$	(supremum)

Here 0 should be neutral for max, successor distributes over max and max is idempotent, subsumptive ($\max(u, u + 1) = u + 1$), associative and commutative. This already generates a non-trivial quotient where for example $\max(0 + 1, u + 1) = u + 1$. To make matters a little more interesting one might add level *metavariables* (noted $?_\ell$) to solve during elaboration

and unification. Then, unification can get stuck: $\max(?_l, ?_k) = \max(?_u, ?_v)$ has no general solution. In Agda and Lean, this requires adding user annotations, instantiating metavariables sufficiently to disambiguate. This situation is quite problematic when one considers automated tactics as used in Rocq or Lean, where it might be difficult to make the tactics annotate generated terms. We would rather like to keep the universe checking entirely automated, so Rocq rather handles a context of universe constraints that are incrementally checked for consistency. Previous versions of Rocq did not run into the problem of getting stuck as the system carefully avoided generating $\max(\_, \_) = \max(\_, \_)$ constraints [4], at the cost of a less expressive theory and unpleasant API: inferred types needed to be "refreshed" before being put into terms, using fresh universe variables as proxies for max expressions. Currently in Rocq, we actually consider a cumulative hierarchy of universes, so equational problems become *entailments* of shape $u_0 \leq \ell_0 \dots u_i \leq \ell_i \vdash u \leq \ell$. Note the restriction to levels on the right hand side of inequality constraints which is enforced by the type inference algorithm [4]. We lift this restriction to levels on the right using the new algorithm of [2], which is able to handle all constraints. *I.e.,* the whole algebra is supported and the entailments to check can now have the general shape: $u_0 \leq u_1 \dots u_i \leq u_i + 1 \vdash u \leq v$. This also greatly simplifies the API for term manipulation. In addition, it cannot "get stuck": it is correct and complete for the (in)equational theory.

The new algorithm [2] is based on a representation of constraints as Horn clauses and performs forward reasoning to check for consistency. Informally, it maintains an assignment of the variables in $\mathbb{N} \cup \{+\infty\}$ that forms a minimal model of the Horn clauses; the value $+\infty$ signals a loop. The algorithm can be used both to check that a new constraint is consistent or produce a loop (used during elaboration), and to decide whether a constraint is deducible from existing constraints (used during kernel type-checking). We extended the algorithm so that each time a constraint $\ell \leq \ell'$ is introduced, we check if $\ell' \leq \ell$ also holds, in which case we choose a canonical variable among the two and substitute the other one by it in the constraints. Then checks for equality between level variables can be done in constant time, which is crucial for performance.

We tested the new implementation against the previous one on the standard library and some large Rocq projects (e.g. math-comp, the HoTT library and Iris), and found a 15% overhead on average, which is expected against a highly optimized implementation for a simpler problem. Note that on average universe checking accounts for 25% of Rocq's compilation times, according to an empirical study by Pierre-Marie Pédrot. However, most libraries do not use polymorphism yet but rely on a very large global graph of constraints, which is less suitable to the new algorithm. On the other hand, experiments on universe polymorphic code shows drastic improvements on performance, with much less constraints being generated and more reasonable inferred quantifications on universes.

Variances

We found the largest slowdowns of the new algorithm in Rocq's `Reals` library, in proof scripts using the `setoid_rewrite` tactics [8]. Those proofs generated a large amount of *global* universe constraints. To solve this, we switched the `setoid_rewrite` tactic to rather use universe polymorphic definitions, avoiding interaction with the global graph. The main polymorphic definitions of interest are:

```
Definition relation@{i} (A : Type@{i}) := A → A → Prop.
Class Proper@{i} {A : Type@{i}} (R : relation@{i} A) (m : A) := R m m.
Definition respectful@{i j} {A : Type@{i}} {B : Type@{j}} (RA : relation A) (RB : relation B)
 : relation (A → B) := fun f g ⇒ ∀ x y, RA x y → RB (f x) (g y).
```

The tactic performs proof search on instances of `Proper@{i} R m`, indexed by the syntax of the relation `R` and the morphism `m`. If done naïvely, however, unification will spend a lot of time unifying universe levels before doing useful work, for example in a unification constraint `relation@{i} A` $\simeq$ `relation@{j} B` will always first try to unify i and j before looking at the arguments `A` and `B`. However one can see that unfolding `relation` would result in a unification that would rather start comparing `A` and `B` as `i` and `j` do not appear in the unfoldings. For `respectful A B RA RB`, the same is true for `i` and `j`.

Hence, we reify information on the occurrences of universe variables as a variance annotation on universe level binders, reusing the variance analysis used for cumulative inductive types. A variance can be one of: irrelevant ($*$), covariant ($+$), contravariant ($-$) or equivariant ($=$). We automatically infer an over-approximation of the variance of each variable at specific positions: in the nth binder type, the return type or the body of the definition. If a variable does not appear in a specific position, it is irrelevant. Typically, parameter universes appear contravariantly in some binder of the definition (e.g. in the 1st binder `A : Type@{i}` for `relation`) but irrelevantly everywhere else. So, if the definition is applied to at least one argument, for example `i` in `relation@{i} A` in `Proper`'s second binder, then we can consider this occurrence irrelevant as well: unfolding `relation@{i} A` makes `i` disappear. Applying this analysis compositionally results in most universe binders being considered as irrelevant in `Proper R m` instances. We can use this automatically inferred information during unification to avoid doing any unification on universes until the real indexes of the proof search (the syntactic shape of the relation and morphism) are unified, resulting in type unifications that will set the universes straight. This improves unification performance significantly. The new `setoid_rewrite` tactic is even more performant than the previous tactic in the `Reals` library.

In addition to improving unification performance, this variance information can also be used to drive the so-called "minimization" algorithm we we now call simplification that solves level metavariables: those appearing in contravariant positions only can be maximized without loss of generality while those appearing in covariant positions only can be minimized without loss of generality. The user can also annotate definitions to provide the exact set of universe variables he expects a definition to quantify over, and simplification will then proceed to instantiate metavariables in terms of the explicitly quantified variables, providing much finer control to the user than the previous implementation. Preliminary experiments using this to develop a new sort-and-universe polymorphic prelude [7, 5] are encouraging.

Acknowledgments We are grateful to Thierry Coquand, Martín Escardó, Pierre-Marie Pédrot & Nicolas Tabareau for helpful discussions on this work.

References

[1] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, December 2015. ISSN 1549-6325. doi: 10.1145/2756553. URL http://doi.acm.org/10.1145/2756553.

[2] Marc Bezem and Thierry Coquand. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theoretical Computer Science*, 913: 1–7, 2022. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2022.01.017. URL https://www.sciencedirect.com/science/article/pii/S0304397522000317.

[3] Robert Harper and Robert Pollack. Type checking with universes. *Theor. Comput. Sci.*, 89(1):107–136, 1991.

[4] Hugo Herbelin. Type Inference with Algebraic Universes in the Calculus of Inductive Constructions. 2005. URL http://pauillac.inria.fr/~herbelin/publis/univalgcci.pdf. Manuscript.

[5] Josselin Poiret, Matthieu Sozeau, and Nicolas Tabareau. Sort and Universe Polymorphic Core Library for Rocq, March 2025. URL https://github.com/jpoiret/coq/tree/new-prelude.

[6] Matthieu Sozeau. Algebraic Universes and Variances for Rocq, March 2025. URL https://github.com/mattam82/coq/tree/universes-clauses.

[7] Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. All Your Base Are Belong to $\mathcal{U}^s$: Sort Polymorphism for Proof Assistants. *Proc. ACM Program. Lang.*, 9(POPL):2253–2281, 2025. doi: 10.1145/3704912. URL https://doi.org/10.1145/3704912.

[8] Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, December 2009.

[9] Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014. ISBN 978-3-319-08969-0. doi: 10.1007/978-3-319-08970-6_32. URL http://dx.doi.org/10.1007/978-3-319-08970-6.

[10] Amin Timany and Matthieu Sozeau. Cumulative Inductive Types In Coq. In Hélène Kirchner, editor, *FSCD*, volume 108 of *LIPIcs*, pages 29:1–29:16, July 2018. ISBN 978-3-95977-077-4. doi: 10.4230/LIPIcs.FSCD.2018.29. URL https://doi.org/10.4230/LIPIcs.FSCD.2018.29.

Internalized Parametricity via Lifting Universals

Aaron Stump

Boston College, Boston, Massachusetts, aaron.stump@bc.edu

Introduction. Internalizing the parametricity principle introduced by Reynolds has several well-known benefits for Type Theory [6], including derivation of induction principles for lambda-encoded data, and "free theorems" derived solely from types [8, 7]. Bernardy and Moulin identified a critical issue taking the (meta-level) relational interpretation $[\![-]\!]$ of a type containing an internalized use $\lceil X \rceil$ of that interpretation [4]: there is a mismatch between the meta-level interpretation, which is indexed by variable renamings, and the internalized version, which it seems should not be. Bernardy and Moulin's solution, based on a symmetry property of relational interpretations, necessitates a pervasive change to the theory, where abstractions now bind hypercubes of variables. Altenkirch et al. follow this geometric approach, leading to a complex formal system [1]. Our goal is a simpler solution.

This abstract describes a constructive type theory $\lceil \mathtt{CC} \rceil$ (pronounced "lift CC") in progress, which extends the Calculus of Constructions (CC) with an internalized parametricity principle. $\lceil \mathtt{CC} \rceil$'s introduces a construct $\lceil A \rceil_i^k$ ("lifting"), with $i \leq k$, where k is the arity of the relation. Definitional equality unfolds applications of this operator, as suggested also by Altenkirch et al. [1]. But $\lceil \mathtt{CC} \rceil$ avoids the mismatch identified by Bernardy and Moulin, by internalizing the meta-level renamings as part of a type form called a *lifting universal*, with syntax $\Pi\, x\langle \bar{x} \rangle : A \,.\, B$.

Syntax. The syntax of $\lceil \mathtt{CC} \rceil$ is shown in Figure 1. It can be seen as an extension of the Calculus of Constructions (CC), and uses the sorts $\star$ and $\Box$ as in [2]. Metavariable θ ranges over the binders. We have generalized forms for Π-types, λ-abstractions, and applications, as well as $\lceil t \rceil_i^k$ for internalized interpretation. For $\lceil t \rceil_i^k$, it is required that $i \leq k$.

Lifting universals. In general, the interpretation of a type T as a k-ary relation is indexed by $k+1$ variable renamings $\rho_0, \cdots, \rho_k$. Bernardy and Lasson realize these renamings by a fixed scheme for deriving the names of new variables $x_0, \cdots, x_{k-1}, \mathring{x}$ from a starting variable x [3]. Here, in contrast, we will consider the renamings explicitly. If $i < k$, then ρ_i maps each free variable x in T to a variable x_i used in speaking about the i'th term that the relation is relating. The renaming ρ_k is used to map $x : R$ to a variable x_k showing that the inputs $\bar{x}$ are related by the interpretation of the type R. For simplicity, we take x_k to be x, so ρ_k is the identity renaming and may be omitted. We write $\bar{x}^k$ for the vector $x_0, \cdots, x_{k-1}$.

$\lceil \mathtt{CC} \rceil$ augments the Π-binder from CC to quantify additionally over $\bar{x}^k$, with syntax $\Pi\, x\langle \bar{x} \rangle : R \,.\, S$. This combines the renamings $x \mapsto x_i$, for $i < k$, with accepting an input x that proves the $\bar{x}$ are related by the relational interpretation of R. The formation, introduction, and elimination

naturals	$\mathbb{N}$	$\ni$	i, j, k		
variables	Var	$\ni$	x, y, z, X, Y, Z		
binders	Bnd	$\ni$	θ	$::=$	$\lambda \mid \Pi$
sorts	Srt	$\ni$	s	$::=$	$\star \mid \Box$
terms	Tm	$\ni$	A, B, C, t, R, S, T	$::=$	$x \mid s \mid \theta\, x\langle \bar{x} \rangle : R \,.\, S \mid t'\, t\langle \bar{t} \rangle \mid \lceil t \rceil_i^k$
contexts	Ctx	$\ni$	Γ	$::=$	$\cdot \mid x\langle \bar{x} \rangle : A, \Gamma$

Figure 1: Syntax for $\lceil \mathtt{CC} \rceil$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x\langle\bar{x}\rangle : A \vdash B : s_2}{\Gamma \vdash \Pi\, x\langle\bar{x}\rangle : A\,.\,B : s_2} \qquad \frac{\Gamma \vdash \Pi\, x\langle\bar{x}\rangle : A\,.\,B : s \quad \Gamma, x\langle\bar{x}\rangle : A \vdash t : B}{\Gamma \vdash \lambda\, x\langle\bar{x}\rangle : A\,.\,t : \Pi\, x\langle\bar{x}\rangle : A\,.\,B}$$

$$\frac{\Gamma \vdash t' : \Pi\, x\langle\bar{x}^k\rangle : A\,.\,C \quad \Gamma \vdash t\langle\bar{t}\rangle : A}{\Gamma \vdash t'\ t\langle\bar{t}^k\rangle : [t\langle\bar{t}\rangle/x]C} \qquad \frac{(\forall\, i < k.\ \Gamma \vdash t_i : [\![A]\!]^k_i) \quad \Gamma \vdash t : [\![A]\!]^k_k\ \bar{t}}{\Gamma \vdash t\langle\bar{t}^k\rangle : A}$$

Figure 2: Typing rules for lifting universals, along with helper judgement $\Gamma \vdash t\langle\bar{t}\rangle : A$

$$\begin{array}{lcll}
[t\langle\bar{t}\rangle/x][\![x]\!]^k_i & = & t_i & i < k \\
[t\langle\bar{t}\rangle/x][\![x]\!]^k_k & = & t & \\
[t\langle\bar{t}^k\rangle/x][\![y]\!]^n_i & = & [\![y]\!]^n_i & n \neq k \ \vee\ y \neq x \\
[t\langle\bar{t}\rangle/x]x & = & t & \\
[t\langle\bar{t}\rangle/x]y & = & y & x \neq y \\
[t\langle\bar{t}\rangle/x]\star & = & \star & \\
[t\langle\bar{t}\rangle/x]\theta\, y\langle\bar{y}\rangle : R\,.\,S & = & \theta\, y\langle\bar{y}\rangle : [t\langle\bar{t}\rangle/x]R\,.\,[t\langle\bar{t}\rangle/x]S & \\
[t\langle\bar{t}\rangle/x](s'\ s\langle\bar{s}\rangle) & = & [t\langle\bar{t}\rangle/x]s'\ [t\langle\bar{t}\rangle/x]s\langle[t\langle\bar{t}\rangle/x]\bar{s}\rangle &
\end{array}$$

Figure 3: Applying a substitution $[t\langle\bar{t}\rangle/x]$ to a term

rules are shown in Figure 2. When $k = 0$, these rules are isomorphic to the usual ones from CC for Π-types, and we use the usual syntax $\Pi\, x : A\,.\,B$ for $\Pi\, x\langle\,\rangle : A\,.\,B$. Similarly, $t'\ t$ abbreviates $t'\ t\langle\,\rangle$, and $\lambda\, x : A\,.\,B$ abbreviates $\lambda\, x\langle\,\rangle : A\,.\,B$. Figure 3 defines substitution. The critical idea is to respect the lifting operator: when substituting into $[\![x]\!]^k_i$, we choose the i'th term from the vector $\bar{t}$ (first equation of Figure 3).

Typing liftings. Figure 4 gives the last typing rules for $[\![\mathtt{CC}]\!]$, which are those for liftings, as well as the axiom $\star : \Box$ and the conversion rule. Parametricity is expressed in rule π. The rules *vr* and *vp* show how assumptions of the form $x\langle\bar{x}\rangle : A$ contribute to typing: x is a proof that $\bar{x}$ is in the relational interpretation of A, and each x_i has type $[\![A]\!]^k_i$, where the lifting is needed to interpret variables y in A introduced with similar assumptions $y\langle\bar{y}\rangle : B$.

Conversion. $[\![\mathtt{CC}]\!]$ uses definitional equality to simplify uses of $[\![-]\!]$. The critical idea is to use a contextual definitional equality of the form $\Gamma \vdash A \simeq B$, and to add an assumption $x\langle\bar{x}\rangle : A$ to the context in the case of a lifting universal. This information may then be used to reduce $[\![x]\!]^k_i$, as expressed in rule L of Figure 5, which says that if we find an assumption $x\langle\bar{x}\rangle : A$ in Γ where the length of $\bar{x}$ is k_j for some j, then we can replace x with x_{i_j}, while retaining the other liftings. (Here, i_j is the requested position from the $[\![x]\!]^{\bar{k}}_{\bar{i}}$ notation.) But this replacement is only allowed for positional liftings, where $i_j < k_j$. This replacement can be justified as a permutation of a positional lifting with other liftings. But such permutation

$$\frac{\Gamma \vdash A : B \quad i < k}{\Gamma \vdash [\![A]\!]^k_i : [\![B]\!]^k_i} \qquad \frac{\Gamma \vdash A : B}{\Gamma \vdash [\![A]\!]^k_k : [\![B]\!]^k_k\ [\![A]\!]^k_0 \cdots [\![A]\!]^k_{k-1}}\ \pi \qquad \frac{}{\Gamma \vdash \star : \Box}$$

$$\frac{x\langle\bar{x}^k\rangle : A \in \Gamma}{\Gamma \vdash x : [\![A]\!]^k_k\ \bar{x}}\ vr \qquad \frac{x\langle\bar{x}^k\rangle : A \in \Gamma \quad i < k}{\Gamma \vdash x_i : [\![A]\!]^k_i}\ vp \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash A \simeq B}{\Gamma \vdash t : B}$$

Figure 4: Typing rules for liftings, plus additional standard rules

$$\frac{x \notin FV(B)}{\Gamma \vdash \Pi\, x\langle \bar{x}^k \rangle : A\,.\,B \simeq \Pi\, \bar{x} : [\![A]\!]^k\,.\,\Pi\, x : [\![A]\!]^k_k\ \bar{x}\,.\,B} \qquad \frac{x\langle \bar{x}^{k_j} \rangle : A \in \Gamma \quad i_j < k_j}{\Gamma \vdash [\![x]\!]^{\bar{k}}_{\bar{i}} \simeq [\![x_{i_j}]\!]^{\bar{k}\setminus k_j}_{\bar{i}\setminus i_j}}\ L$$

$$\frac{S \equiv \Pi\, X : A\,.\,B}{\Gamma \vdash [\![S]\!]^k_k \simeq \lambda\, \bar{Y}^k : S\,.\,\Pi\, X\langle \bar{X}^k \rangle : A\,.\,[\![B]\!]^k_k\ (\bar{Y}\ \bar{X})} \qquad \frac{}{\Gamma \vdash [\![A\ B]\!]^k_k \simeq [\![A]\!]^k_k\ [\![B]\!]^k_0 \cdots [\![B]\!]^k_{k-1}}$$

$$\frac{\Gamma \vdash A \simeq A' \quad \Gamma, x\langle \bar{x} \rangle : A \vdash B \simeq B'}{\Gamma \vdash \theta\, x\langle \bar{x} \rangle : A\,.\,B \simeq \theta\, x\langle \bar{x} \rangle : A'\,.\,B'} \qquad \frac{i < k}{\Gamma \vdash [\![\star]\!]^k_i \simeq \star} \qquad \frac{}{\Gamma \vdash [\![\star]\!]^k_k \simeq \lambda\, \bar{Y}^k : \star\,.\,\bar{Y} \to \star}$$

Figure 5: Selected rules for definitional equality.

does not make sense for relational liftings ($[\![A]\!]^k_k$), where arity-k and arity-j relational liftings result in different arity relations, and hence could not be permuted. The first rule of Figure 5 makes $\Pi\, x\langle \bar{x} \rangle : A\,.\,B$ an abbreviation for the nested Π-type one would expect from the relational interpretation of $\Pi\, x : A\,.\,B$, as long as x has been completely eliminated from the body. The middle row of Figure 5 expresses the relational semantics of Π-types using lifting universals, and applications using positional liftings ($0 \leq i < k$) and the relational lifting ($i = k$). Positional lifting behaves homomorphically with respect to the constructs of CC (rules omitted).

Example: iterated internal parametricity. Internalized unary parametricity is expressed as $\Pi\, A : \star\,.\,\Pi\, a : [\![A]\!]^1_0\,.\,[\![A]\!]^1_1\ a$ (abbreviate this $\mathcal{T}$). The type given for a is as required by the type of $[\![A]\!]^1_1$, based on rule π. Let us calculate $[\![\mathcal{T}]\!]^2_2$.

$$\begin{array}{ll} [\![\mathcal{T}]\!]^2_2 & \simeq \\ \lambda\, \bar{X}^2 : \mathcal{T}\,.\,\Pi\, A\langle \bar{A}^3 \rangle : \star\,.\,\Pi\, a\langle \bar{a}^3 \rangle : [\![A]\!]^1_0\,.\,[\![[\![A]\!]^1_1\ a]\!]^2_2\ (\bar{X}\ \bar{A}\ \bar{a}) & \simeq \\ \lambda\, \bar{X}^2 : \mathcal{T}\,.\,\Pi\, A\langle \bar{A}^3 \rangle : \star\,.\,\Pi\, a\langle \bar{a}^3 \rangle : [\![A]\!]^1_0\,.\,[\![[\![A]\!]^1_1]\!]^2_2\ [\![a]\!]^2_0\ [\![a]\!]^2_1\ (\bar{X}\ \bar{A}\ \bar{a}) & \simeq \\ \lambda\, \bar{X}^2 : \mathcal{T}\,.\,\Pi\, A\langle \bar{A}^3 \rangle : \star\,.\,\Pi\, a\langle \bar{a}^3 \rangle : [\![A]\!]^1_0\,.\,[\![[\![A]\!]^1_1]\!]^2_2\ a_0\ a_1\ (X_0\ A_0\ a_0)\ (X_1\ A_1\ a_1) & \end{array}$$

Call the last type above Q, and let us see how its body is typable. For readability, write $\mathcal{A}_i$ for $[\![A]\!]^1_i$. $\mathcal{A}_i$ does not equal A_i, as the length of $\bar{A}$ in the context is 3, which does not match the arity 1 of this lifting. The type of $[\![[\![A]\!]^1_1]\!]^2_2$ is the following, again followed by several definitionally equal types:

$$\begin{array}{ll} [\![[\![\star]\!]^1_1\ \mathcal{A}_0]\!]^2_2\ [\![\mathcal{A}_1]\!]^2_0\ [\![\mathcal{A}_1]\!]^2_1 & \simeq \\ [\![\mathcal{A}_0 \to \star]\!]^2_2\ [\![\mathcal{A}_1]\!]^2_0\ [\![\mathcal{A}_1]\!]^2_1 & \simeq \\ \Pi\, x\langle \bar{x}^3 \rangle : \mathcal{A}_0\,.\,[\![\mathcal{A}_1]\!]^2_0\ x_0 \to [\![\mathcal{A}_1]\!]^2_1\ x_1 \to \star & \end{array}$$

To type the body of Q, we use rule L to simplify a positional lifting when it is nested in another lifting. For one example, using L, we can prove that the type of $(X_0\ A_0\ a_0)$, which is $[\![A_0]\!]^1_1\ a_0$, equals $[\![[\![A]\!]^1_1]\!]^2_0\ a_0$. Since we have $A(\bar{A}^3) : \star$ in the context, the positional lifting $[\![[\![A]\!]^1_1]\!]^2_0$ is equal to $[\![A_0]\!]^1_1$, replacing A with A_0. This holds similarly for the types of $\bar{a}$ and $(X_1\ A_1\ a_1)$, allowing the body of Q to be typed.

Towards normalization. Developing a semantics for $[\![\texttt{CC}]\!]$ seems challenging, because of liftings. One would need to define a relational semantics that can be iteratively applied, so one could apply the semantics to an object already in the semantic domain. Instead of this path, I propose to use the Girard projection to reduce normalization of $[\![\texttt{CC}]\!]$ to normalization of F_ω, as in [2]. It then becomes plausible to consider internalizing Girard projection as a type construct, as in [5], which would allow new definitional equalities as discussed in [8].

Acknowledgments. Thanks to the anonymous reviewers for helpful comments.

References

[1] Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. Internal parametricity, without an interval. *Proc. ACM Program. Lang.*, 8(POPL):2340–2369, 2024.

[2] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford University Press, 12 1992.

[3] Jean-Philippe Bernardy and Marc Lasson. Realizability and parametricity in pure type systems. In Martin Hofmann, editor, *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*, volume 6604 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2011.

[4] Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 135–144. IEEE Computer Society, 2012.

[5] Jean-Philippe Bernardy and Guilhem Moulin. Type-theory in color. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 61–72, New York, NY, USA, 2013. Association for Computing Machinery.

[6] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[7] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.

[8] Philip Wadler. The girard-reynolds isomorphism (second edition). *Theor. Comput. Sci.*, 375(1-3):201–226, 2007.

Extending Sort Polymorphism with Elimination Constraints

Tomás Díaz[1], Johann Rosain[2], Matthieu Sozeau[2], Nicolas Tabareau[2], and Théo Winterhalter[3]

[1] University of Chile, Chile
[2] LS2N & Inria de l'Université de Rennes, Nantes, France
[3] LMF & Inria Saclay, Saclay, France

Sort Polymorphism [1] provides a new axis for parameterizing definitions in dependent type theory by sorts, extending universe polymorphism which allows parameterization by universe levels and, in cumulative systems, universe constraints modeling predicative universe inclusion. In this talk, we will present a refinement of sort polymorphism with a different kind of constraints to model the elimination rules from one sort to another. These constraints can model the existing sort elimination constraints used in the Rocq Prover to specify the interactions between: an impredicative, definitionally proof-irrelevant `SProp` sort, an impredicative `Prop` sort with so-called "(sub)singleton" elimination, and a predicative and cumulative `Type` hierarchy. Using bounded quantification on sorts with elimination constraints, we can derive a single, most-general induction principle on each inductive type, reducing code duplication, and give an elegant treatment to primitive record types and their projections. We will demonstrate the usefulness of this generalization, on existing sorts and new ones. During the talk, we will discuss the state of the implementation in the Rocq Prover and its formal verification in the MetaRocq project.

The Need for Sort Elimination Constraints

Sort polymorphism [1] has been integrated in the Rocq Prover since version 8.19 [4]; see the Reference Manual for an introduction. It basically provides a way to parameterize over *sort quality variables* in addition to universe level variables in definitions. A sort is then thought of as a pair of a sort quality and a universe level. This enables the definition of generic connectives like the unit type/true proposition, the empty type/absurd proposition, the Cartesian product/conjunction and sums/disjunctions, as unique inductive types, interpretable in various sorts. For example sums can be defined as:

```
Inductive sum@{sl sr s | ul ur} (A : 𝒰@{sl | ul}) (B : 𝒰@{sr | ur}) : 𝒰@{s | max(ul,ur)} :=
| inl : A → sum A B
| inr : B → sum A B
```

Herein, `sum` quantifies over sort qualities `sl` and `sr` for its parameters, and over sort `s` for its return type. It can thus be instantiated to the usual sum type `A + B` by using `Type` everywhere, to disjunction `A ∨ B` in either `Prop` or `SProp`, or to a combination of those, mixing propositional and computational content:

```
Definition decidable@{i} (A : Prop) : Type@{i} := sum@{Prop Prop Type | 0 i} A (¬ A).
Definition type_excluded_middle@{i} (A : Type@{i}) : Prop := sum@{Type Type Prop | i i 0} A (¬ A).
```

Sort polymorphism introduces a simple rule to determine which eliminations are allowed for a sort polymorphic inductive type: by default only elimination for motives in the *same sort*

quality as the inductive's type quality is allowed. For sums, this means the following constant is derived:

```
sum_elim@{sl sr s| ul ur i} A B {P : sum A B → 𝒰@{s | i}} : (∀ a, P (inl a)) → (∀ b, P (inr b)) → ∀ s, P s.
```

This models in particular that `type_excluded_middle A B` may only be eliminated to `Prop`, respecting the elimination constraints of Rocq (`Prop` cannot be eliminated to `Type` except in the particular case of singleton types). However, it sadly forbids elimination of `A + B`, which lives in `Type`, into `Prop` or `SProp`, even though it is entirely sound: one can still explicitly define an eliminator from `A + B` to `Prop`.

To remedy this situation, we propose to enable the definition of a single eliminator whose instantiations are restricted depending on a set of *sort elimination constraints*. They are of the form `s → s'`, meaning that inductive types in sort `s` can safely be eliminated to sort `s'`. This leads to `sum_elim` below.

```
Definition sum_elim@{sl sr s s' | ul ur u' | s → s'} A B (P : sum@{sl sr s| ul ur} A B → 𝒰@{s' | u'})
  (fl : ∀ a, P (inl a)) (fr : ∀ b, P (inr b)) x : P x :=
  match x with inl a ⇒ fl a | inr b ⇒ fr b end.
```

Theory

We propose to define the *sort elimination relation*, denoted $\rightarrow$, as the reflexive and transitive closure of a set of elimination constraints, defining a partial order. The motivations for such a behavior are twofold: (i) it allows the user to define only the elimination constraints that are strictly necessary, and (ii) it enables the checking mechanism to easily find inconsistencies, e.g. in the case where the user wants a sort `s` such that `SProp → s` and `s → Type`.

Moreover, we postulate that giving an antisymmetric structure to this relation is sound, and even quite helpful: having two sorts `s` and `s'`, such that `s → s'` and `s' → s`, means that the same types (up to isomorphism) inhabit the two sorts: we do not actually want to differentiate them.

These properties make elimination constraints akin to universe level constraints [3], and allow us to reuse the same data-structure for both kinds of constraints: an acyclic graph. One still has to be careful to add the *ground elimination constraints* `Type → Prop → SProp`, something that was not needed for universe level constraints but which is necessary here to forbid equating e.g. `Prop` and `Type` due to user-imposed elimination constraints.

Implementation

The implementation of elimination constraints in Rocq was also the opportunity to refactor and harmonize the internal handling of elimination. Indeed, previously, sort-quality-related logic was scattered across multiple files, with ad-hoc implementations, duplication and an unclear separation of concerns. For instance, some parts of the code used direct sort quality comparisons, while others used set membership checks or variables relevance to validate eliminations between sorts. In addition, these were often mixed with orthogonal universe level checks.

To address this, we refactored the implementation into two dedicated modules: the existing `Sort` module, responsible for basic sort quality manipulation, and a new quality elimination constraint graph, which gathers elimination rules and ensures consistency through an acyclicity check.

On top of that, we also cleaned-up the full elimination mechanism implemented in Rocq. It is based on a finer-grained principle than the acyclicity check, and enables a more subtle management of elimination on a `match` on an inductive `I` in sort `s` when the motive has sort `s'`.

The following table summarizes it: Here, by instantiating `s` by `SProp`, `Prop` or `Type`, it is clear that

#constructors(`I`)	Elimination constraint between `s` and `s'`
0	`s` → `SProp` or `s` → `s'`
1	(`sort(arg)` → `s` for every arg and `s` → `Prop`) or `s` → `s'`
2 or more	`s` → `s'`

one finds back the expected behavior: zero-constructor inductives in `SProp`/`Prop`/`Type` eliminate to any sort, while singleton inductives in `Prop` and `Type` have no elimination restriction.

Nevertheless, a special case persists in the form of elimination in fixpoints, which is still managed in an ad-hoc way: only the acyclicity check is used, together with a rule that always allows fixpoints on inductives in `Prop`. A useful instance of this occurs on fixpoints on accessibility proofs `Acc` (when using its projector `Acc_inv`).

Verification

As our design is now stabilizing, we are aiming at a mechanized formal model of elimination constraints and the new elimination checks along with sort polymorphism in the MetaRocq framework [2], which currently only handles `Prop` and `Type` with (sub)singleton elimination. While parameterizing definitions with sorts and (proof-irrelevant) elimination constraints should be a straightforward extension of bounded universe polymorphism, adapting the definition and formalization of erasure and its correctness proof will be an interesting challenge, we hope to report on this by the time of the conference.

Acknowledgments We are thankful to Pierre-Marie Pédrot and Gaëtan Gilbert for insightful discussions on the sort polymorphism and elimination restriction implementations in the Rocq Prover.

References

[1] Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. *All Your Base Are Belong to $\mathcal{U}^s$: Sort Polymorphism for Proof Assistants*. *Proc. ACM Program. Lang.*, 9(POPL):2253–2281, 2025.

[2] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. *Correct and Complete Type Checking and Certified Erasure for Coq, in Coq*. *J. ACM*, 72(1), January 2025. ISSN 0004-5411.

[3] Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014.

[4] The Coq Development Team. The Coq Proof Assistant, version *8.19*. June 2024.

The Case for Impredicative Universe Polymorphism

Stefan Monnier[1]

Université de Montréal - DIRO, Montréal, Canada
monnier@iro.umontreal.ca

Abstract

Predicativity tells us that universe polymorphic definitions should live in the universe of level ω. We argue for an alternative rule we call IUP (impredicative universe polymorphism) which allows such definitions to live in a much lower universe. The conditions under which this rule leads to inconsistencies are not yet known, so in the mean time we present different use cases to provide some intuition about its possibilities and its limits.

1 Introduction

The predicative tower of universes is nowadays ubiquitous in proof assistants, and more often than not it comes equipped with some form of universe polymorphism [2]. This can take various forms, from a "simple" sort of implicit meta-level macro-expansion, through *displacement algebras* [5], to something like Agda where universe levels can be manipulated explicitly and universe polymorphic definitions can be manipulated as first-class values [10], or even systems where universe levels are themselves first-class values [8].

In all these works, universe polymorphism is predicative, meaning that universe polymorphic definitions are placed in a universe of level strictly above the largest level to which it can be instantiated. Concretely, this typically means that such definitions are placed in the universe of level ω.

We argue that, under some conditions, we could place universe polymorphic definitions in a lower universe, making those definitions impredicative. More specifically, for a type τ of universe level ℓ, which can depend on a universe level variable l, the accepted predicative rule places the type $\forall l.\tau$ in the universe of level $\mathsf{sup}_l\ \ell$, i.e. ω, whereas we argue that, under some conditions, we could place it in the universe $\mathsf{inf}_l\ \ell$, which in our case is also $\ell[0/l]$.

In terms of practical uses, this form of universe polymorphism would be convenient to manipulate universe-polymorphic terms without resorting to universe levels beyond ω, it could be used as an alternative form of impredicativity which does not require a special impredicative universe, and it appears to be one of the ingredients necessary for type-preserving closure conversion of languages with a hierarchy of universes.

We do not know under which conditions we can use such a rule without breaking consistency of the system, and it seems difficult to relate this form of impredicativity we call IUP [9, Sec 4] to other known forms of impredicativity such the Prop universe of the Calculus of Constructions [4] or the propositional resizing axiom of HoTT [11, Sec 3.5], so until the underlying theory is better understood, and as a way to motivate such investigation, we present a few specific use cases where we have reasons to think they should be accepted or not.

2 Case Studies

We explore a few use cases, chosen for their ability to provide some intuition about the possibilities and the limits of IUP.

2.1 Strong sums

One clear limit to the consistency of IUP comes if we apply it to strong sums, as in:

$$\frac{\Gamma, l : \mathsf{Level} \vdash \tau \ : \ \mathsf{Type}_\ell}{\Gamma \vdash \Sigma l.\tau \ : \ \mathsf{Type}_{\ell[0/l]}}$$

Since we can then resize any function down to universe Type_1 by stuffing it in an object of type $\Sigma l_1.\Sigma t_1 : \mathsf{Type}_{l_1}.\Sigma l_2.\Sigma t_2 : \mathsf{Type}_{l_2}.t_1 \to t_2$ out of which we can easily recover the original function, regardless of its original universe level. This also suggests that IUP is incompatible with first-class universe levels.

2.2 Church encoding

Functions that correspond to Church encodings of inductive types are examples where the IUP rule seems to be sound. Consider a type of the form:

$$\forall l.\forall t : \mathsf{Type}_l.t \to (\tau \to t \to t) \to t$$

Assuming that τ do not refer to l nor t, this type corresponds to a Church-style encoding of a list of elements of type τ. If τ is in universe level ℓ, then the corresponding inductive type would be placed in universe level ℓ, but Agda would place it in ω. The IUP rule would instead put the above type in universe $1 \sqcup \ell$. Compared to the original Church encoding, this adds support for strong elimination, i.e. the possibility to eliminate to any universe rather than only to the bottom impredicative universe (i.e. Prop or Set), which is allowed for inductive types and thus suggests the IUP rule should be sound in this case.

More generally, we can consider a type of the following form:

$$\forall l.\forall t : \mathsf{Type}_l.(\vec{\tau_1} \to t) \to ... \to (\vec{\tau_n} \to t) \to t$$

If we assume that the τ_{ij} types do not refer to l, this type corresponds to a Church-style encoding of an inductive type. If each τ_{ij} is in universe level ℓ_{ij}, then the corresponding inductive type would be placed in universe level $\bigsqcup_{i,j} \ell_{ij}$, while our IUP rule would similarly put the above type in universe $1 \sqcup \bigsqcup_{i,j} \ell_{ij}$.
Note that compared to actual inductive types, Church-style encodings using our IUP rule still lack dependent elimination, which is a problem that has been tackled by Awodey et.al. [1] and Firsov and Stump [6]. Of mention also would be the work by Jenkins et.al. [7] which tries to support strong elimination but without a rule like our IUP rule.

2.3 Closures

In type-preserving compilers for functional programming languages, functions end up reified as tuples that are then wrapped in an existential package so as not to expose the set of captured variables in its type. When captured variables can come from an arbitrary universe level, we similarly need to hide that level: For a source function of type $\tau_1 \to \tau_2$, the closure needs to

have a type of the form $\exists l.\exists t : \mathsf{Type}_l.t \times ((\tau_1 \times t) \to \tau_2)$ where t is the type of the captured environment. If those $\exists$ are weak sums, one cannot do anything with such closures that one cannot already do with its corresponding function of type $\tau_1 \to \tau_2$, so it supports the idea that such existentials could live in the same universe level $l_1 \sqcup l_2$, and thus suggests that our IUP rule is justified in this case to puts this existential type in universe level $1 \sqcup l_1 \sqcup l_2$.
See Bowman and Ahmed [3] for more challenges with type preserving closure conversion.

2.4 Well-ordering

Girard's paradox starts with an impredicative definition of an ordering and then exhibits an ordering of such orderings. The ordering is a tuple of a type along with some properties:

$$\mathsf{type}\ \mathit{Ordering} : \mathsf{Type} = \mathit{mk\text{-}ord}\ (\mathit{set} : \mathsf{Type})\ (\mathit{less\text{-}than} : \mathit{set} \to \mathit{set} \to \mathsf{Type})\ ...$$

In a predicative setting this does not work because *Ordering* ends up in a higher universe level than *set*. If we want to try and reproduce the paradox using IUP, we need to abstract over the universe level of *set*:

$$\mathsf{type}\ \mathit{Ordering} : \mathsf{Type}_1 = \mathit{mk\text{-}ord}\ (l : \mathsf{Level})\ (\mathit{set} : \mathsf{Type}_l)\ (\mathit{less\text{-}than} : \mathit{set} \to \mathit{set} \to \mathsf{Type}_l)\ ...$$

But since IUP is restricted to apply to weak sums only, the above definition is unusable, because the *set* cannot be extracted from that tuple any more, making it difficult to compare two such orderings to make an ordering of orderings.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant N^o RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

References

[1] Steve Awodey, Jonas Frey, and Sam Speight. Impredicative encodings of (higher) inductive types. In *Annual Symposium on Logic in Computer Science*, page 17, 2018. doi:10.1145/3209108.3209130.

[2] Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. Type theory with explicit universe polymorphism. In *Types for Proofs and Programs*, Leibniz International Proceedings in Informatics (LIPIcs), 2022. doi:10.4230/LIPIcs.TYPES.2022.13.

[3] William J. Bowman and Amal Ahmed. Typed closure conversion for the calculus of constructions. In *Programming Languages Design and Implementation*, page 797–811, 2018. doi:10.1145/3192366.3192372.

[4] Thierry Coquand and Gérard P. Huet. The calculus of constructions. Technical Report RR-0530, INRIA, 1986.

[5] Kuen-Bang Hou (Favonia), Carlo Angiuli, and Reed Mullanix. An order-theoretic analysis of universe polymorphism. In *Principles of Programming Languages*, pages 1659–1685. ACM Press, 2023. doi:10.1145/3571250.

[6] Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in Cedille. In *Certified Programs and Proofs*, pages 215–227, 2018. doi:10.1145/3167087.

[7] Christa Jenkins, Andrew Marmaduke, and Aaron Stump. Simulating large eliminations in Cedille. In *Types for Proofs and Programs*, Leibniz International Proceedings in Informatics (LIPIcs), page 22, 2021. doi:10.4230/LIPIcs.TYPES.2021.9.

[8] András Kovács. Generalized universe hierarchies and first-class universe levels. In *Computer Science Logic*, pages 28:1–28:17, 2022. doi:10.4230/LIPIcs.CSL.2022.28.

[9] Stefan Monnier and Nathaniel Bos. Is impredicativity implicitly implicit? In *Types for Proofs and Programs*, Leibniz International Proceedings in Informatics (LIPIcs), pages 9:1–9:19, 2019. doi:10.4230/LIPIcs.TYPES.2019.9.

[10] The Agda team. The Agda manual. URL: https://agda.readthedocs.io/en/v2.6.2.1/.

[11] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: https://arxiv.org/abs/1308.0729.

From parametricity to identity types

Towards Higher Observational Type Theory

Thorsten Altenkirch[1], Ambrus Kaposi[2]*, Michael Shulman[3]†, and Elif Üsküplü[4]

[1] University of Nottingham, Nottingham, United Kingdom, txa@cs.nott.ac.uk
[2] Eötvös Loránd University, Budapest, Hungary, akaposi@inf.elte.hu
[3] University of San Diego, San Diego, USA, shulman@sandiego.edu
[4] Indiana University, Bloomington, IN, USA, euskuplu@iu.edu

We have introduced Parametric Observational Type Theory, which is a theory with internal parametricity which does not use an interval. It satisfies canonicity [2]; see [4] for an indexed version. This theory is the basis for the Narya system that we have implemented.[1] We are planning to use this type theory as a stepping stone to Higher Observational Type Theory (HOTT), a computational version of homotopy type theory that likewise does not use an interval. POTT introduces a relation for every type (called the *bridge type* [6]). The bridge type (which we will write as `Br` in Narya) can be seen as a baby version of equality: it is reflexive and a congruence, but it is not symmetric or transitive. Bridge can be heterogeneous, so given `A : B → Type`, we have `Br A : (b0 b1 : B) → Br B b0 b1 → A b0 → A b1 → Type`. Bridge of Π types says that bridge-identical inputs are mapped to bridge-identical outputs. The bridge of an inductive type is an inductive family with the same number of constructors, as usual in parametricity translations [5], and symmetrically for coinductive types. Bridge of the universe corresponds to relation space; in particular, if `A2 : Br Type A0 A1`, and `a0 : A0` and `a1 : A1`, then `A2 a0 a1 : Type`. Narya also supports Martin-Löf's inductively defined equality type family (which does not satisfy function extensionality or univalence).

Following the ideas of [8, 3] we want to construct a universe of fibrant types inside POTT, such that bridges of fibrant types correspond to equivalences (i.e. we have univalence), and using the observation that singletons are contractible we can derive the usual J-elimination rule (with weak β-equality). This is similar to the approach to cubical type theory in [7].

Our goal is to define a family `isFib : Type → Type` in Narya; then the universe of fibrant types will be `Ufib = Σ Type isFib`. HOTT will live in `Ufib`, and Narya can be seen as an outer layer for a two-level type theory [1]. We define `isFib` as a 'higher coinductive type'. Higher coinductive types are the dual of higher inductive types, and come with destructors on equalities rather than equality-constructors. In POTT and Narya, these are actually bridge-destructors. The Narya-definition of `isFib` is the following; note the 'self variable' `x` through which previous fields are accessed.

```
def isFib (A : Type) : Type := codata [
  | x .trr.p   : A.0 → A.1
  | x .trl.p   : A.1 → A.0
  | x .liftr.p : (a0 : A.0) → A.2 a0 (x.2 .trr a0)
  | x .liftl.p : (a1 : A.1) → A.2 (x.2 .trl a1) a1
  | x .id.p    : (a0 : A.0) (a1 : A.1) → isFib (A.2 a0 a1) ]
```

Destructors in Narya start with a `.`, and higher destructors end with `.p` (our chosen name for the parametricity direction) when being defined, but not when they are used. For example, given

*Ambrus Kaposi was supported by the the HOTT ERC Grant 101170308.
†Michael Shulman was supported by the United States Air Force Office of Scientific Research under award number FA9550-21-1-0009.

[1] See https://github.com/gwaithimirdain/narya/tree/2cca3d7 and more specific links in the abstract.

```
A0 : Type,                  x0 : isFib A0,
A1 : Type,                  x1 : isFib A1,
A2 : Br Type A0 A1,         x2 : Br isFib A0 A1 A2 x0 x1,
```

we have `x2 .trr : A0 → A1`. Also, since a higher destructor must be applied to a *bridge in* the type being defined, the 'self variable' `x` and the parameter `A` must be replaced by such bridges when declaring the its type; the suffixes `.0`, `.1`, `.2` on the variables `A` and `x` serve to identify the three components of these arguments of the heterogeneous bridge types.

`isFib` expresses that given two bridge-identical types, we can transport between them in both directions (via `.trr` and `.trl`), these transports preserve the bridge-relation `A.2` (via `.liftr` and `.liftl`), and finally, the bridge-relation itself is fibrant (via `.id`; this last destructor makes `isFib` into a coinductive type rather than just a record).

Higher coinductive types are terminal coalgebras; in Narya we can construct their elements via co-pattern matching. Similarly, the bridge of a higher coinductive type is another one. This allows us to prove that a fibrant one-to-one correspondence between fibrant types, i.e. an equivalence, gives rise to a bridge of those types; thus we have univalence. Univalence is *definitional*, which means that if we turn a function which is an equivalence into an element of the `Br` type, and transport along it, we get back the function definitionally (unlike in cubical type theories such as Cubical Agda).

We can also prove using co-pattern matching and corecursion that basic type formers preserve fibrancy. Fibrancy for the empty type is trivial using an empty match. For the unit type, we show that bridge of unit is equivalent to unit, with the notion of equivalence using the Martin-Löf equality type. Knowing that fibrancy is preserved by such equivalences, we prove fibrancy of unit via corecursion. For natural numbers, we first show that bridge is equivalent to the recursively defined equality (which only uses empty and unit, already shown to be fibrant). Then fibrancy of `Nat` is a simple corecursion. Fibrancy of Σ-types, Π-types, W-types, and M-types can be shown with analogous methods, although we need to assume as axioms that the Martin-Löf equality type satisfies function extensionality (for W-types) and coinductive extensionality (for M-types). The more serious challenge of showing that `Ufib` itself is fibrant is work in progress.

We shouldn't forget the elephant in the room, which is a semantic justification of higher coinductive types in POTT and an extension of the proof of canonicity to their presence. We also aim to extend the proof of canonicity to a proof of normalization, based on the algorithm implemented by Narya. Work is also in progress to directly implement a mode for HOTT in Narya, in which all types will automatically be fibrant.

The above results were formalised in Narya by the third author. Note that Narya currently does not have termination or universe-level checking; thus while our corecursive proofs of fibrancy appear semantically productive, they have not been syntactically checked to be so.

References

[1] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending homotopy type theory with strict equality. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, volume 62 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. URL: https://doi.org/10.4230/LIPIcs.CSL.2016.21, doi:10.4230/LIPICS.CSL.2016.21.

[2] Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. Internal parametricity, without an interval. *Proc. ACM Program. Lang.*, 8(POPL):2340–2369, 2024. doi:10.1145/3632920.

[3] Thorsten Altenkirch and Ambrus Kaposi. Towards a cubical type theory without an interval. In *21st International Conference on Types for Proofs and Programs (TYPES 2015)(2018)*, pages 3–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.

[4] Thorsten Altenkirch, Ambrus Kaposi, Michael Shulman, and Elif Usküplü. Internal relational parametricity, without an interval. In *30th International Conference on Types for Proofs and Programs TYPES 2024–Abstracts*, page 56, 2024.

[5] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012. doi:10.1017/S0956796812000056.

[6] Evan Cavallo. *Higher Inductive Types and Internal Parametricity for Cubical Type Theory.* PhD thesis, Carnegie Mellon University, USA, 2021.

[7] Ian Orton and Andrew M. Pitts. Axioms for modelling cubical type theory in a topos. *Logical Methods in Computer Science*, 14, December 2018. arXiv:1712.04864. doi:10.23638/LMCS-14(4:23)2018.

[8] Michael Shulman. Towards third-generation HoTT, part 1, 2021. YouTube lecture. URL: https://www.youtube.com/watch?v=FrxkVzItMzA.

About the construction of simplicial and cubical sets in indexed form: the case of degeneracies

Hugo Herbelin[1] and Ramkumar Ramachandra[2]

[1] INRIA - CNRS - IRIF - Université Paris Cité
[2] unaffiliated

Abstract

We presented in [HR25] a parametricity-based construction of augmented semi-simplicial and semi-cubical sets in indexed form. Since then, we refined the construction in two directions: the addition of degeneracies and a fine-grained analysis of dependencies allowing to replace some propositional equalities by definitional ones.

The now classical problem of defining semi-simplicial types in type theory[1] popularised the idea that semi-simplicial sets could alternatively be built in an "indexed way" as the following coinductive family of family of sets (here for the augmented case):

$$\begin{array}{lcl} X_{-1} & : & \mathsf{HSet} \\ X_0 & : & X_{-1} \to \mathsf{HSet} \\ X_1 & : & \Pi x : X_{-1}.\, X_0(x) \times X_0(x) \to \mathsf{HSet} \\ X_2 & : & \Pi x : X_{-1}.\, \Pi yzw : X_0(x).\, X_1(x)(y,z) \times X_1(x)(y,w) \times X_1(x)(z,w) \to \mathsf{HSet} \\ \multicolumn{3}{c}{\ldots} \end{array}$$

This amounts to see a presheaf on a direct category as a sequence of sets fibred one over the others and to iteratively apply the fibred-indexed correspondence[2]:

$$\text{(fibred)} \qquad (\Sigma T : \mathsf{HSet}.(T \to S)) \;\simeq\; (S \to \mathsf{HSet}) \qquad \text{(indexed)}$$

Furthermore, the category of augmented semi-simplices and the category of cubes, and thus augmented semi-simplicial sets and cubical sets as well, can uniformly be described as categories over ω whose morphisms between n and p are the words of length p on an alphabet $L \uplus \{0\}$ containing n times the letter 0, where L is the one-letter set $\{+\}$ in the augmented semi-simplex case and the two-letter set $\{+,-\}$ in the semi-cube case. Such description is similar to iterating Reynolds parametricity [Rey72], resulting in [HR25, Tables 1, 2, 3, 4, 5] to a uniform definition of augmented semi-simplicial and semi-cubical sets, that is, in the first case, of a family X_{-1}, X_0, X_1, ... as above. It relies on reformulating the type of each X_n under the form $\mathsf{frame}^n(X_{-1},...,X_{n-1}) \to \mathsf{HSet}$ where frame^n, defined recursively, decomposes the border of a simplex/cube into n layers made of appropriate filled simplices/cubes (we will write $\mathsf{frame}^{n,p}$ for the prefix of frame^n made of the p first layers; we will also write $\mathsf{restrframe}_q^{n,p}(d)$ for an auxiliary operation of the construction used to project along direction q a partial $d : \mathsf{frame}^{n+1,p}(X_{-1},...,X_n)$ into a $\mathsf{frame}^{n,p}(X_{-1},...,X_{n-1})$).

Adding degeneracies. The indexed construction of augmented semi-simplicial and semi-cubical sets can be equipped with degeneracies by asserting the existence of maps from X_n

[1] ncatlab.org/nlab/show/semi-simplicial+types+in+homotopy+type+theory

[2] A practical interest of such presentation of simplicial sets or cubical sets is to provide models of type theory that preserve the indexed form of dependent types, and thus liable to eventually interpret bridges or univalence definitionally.

to X_{n+1}, each applied to appropriate arguments reflecting the coherence conditions between degeneracies and faces. The degeneracies we are considering are those of usual (binary) cubical sets and of unary cubical sets as those found in parametric type theory [BCM15] (note that, along the above uniform description of augmented simplicial sets and cubical sets, degeneracies in simplicial sets are actually the unary case not of the degeneracies of cubical sets but of the connections of cubical sets!). Moreover, we consider also only one degeneracy, namely the one in the last direction of a simplicial or cubical shape (the other degeneracies could eventually be obtained by adding permutations to the structure). For a given frame d and $w : X_n(d)$, the degeneracy in the last direction $r_n(d)(w)$ has to lay on an appropriate frame for X_{n+1} whose last component is w. For instance, in the first dimensions, it takes the form:

$$\begin{array}{lcl} r_{-1} & : & \Pi x : X_{-1}.\, X_0(x) \\ r_0(x) & : & \Pi y : X_0(x).\, X_1(x)(r_{-1}(x), y) \\ r_1(x)(y,z) & : & \Pi w : X_1(x)(y,z).\, X_2(x)(r_{-1}(x), y, z)(r_0(x)(y), r_0(x)(z), w) \end{array}$$

Each r_n depends on the previous ones, so, given a sequence $(X_{-1}, X_0, ...)$ characterised in [HR25] by a coinductively-defined type $\nu\mathsf{Set}$, we coinductively define an infinitely nested Σ-type $\nu\mathsf{reflSet}$ representing the type of sequences $r_{-1}, r_0, r_1, ...$ as follows:

$$\begin{array}{l} \nu\mathsf{reflSet}(X_{-1}, X_0, ...) \triangleq \\ \quad \Sigma r_{-1} : \Pi d : \mathsf{frame}^{-1}.\, \Pi x : X_{-1}(d).\, X_0(\mathsf{reflframe}^{-1}(d), x). \\ \quad \Sigma r_0 : \Pi d : \mathsf{frame}^{0}(X_{-1}).\, \Pi x : X_0(d).\, X_1(\mathsf{reflframe}^{0}(r_{-1})(d), x). \\ \quad \Sigma r_1 : \Pi d : \mathsf{frame}^{1}(X_{-1}, X_0).\, \Pi x : X_1(d).\, X_2(\mathsf{reflframe}^{1}(r_{-1}, r_0)(d), x). \\ \quad ... \end{array}$$

where

$$\mathsf{reflframe}^n(r_{-1}, ..., r_{n-1}) : \mathsf{frame}^n(X_{-1}, ..., X_{n-1}) \to \mathsf{frame}^{n+1,n}(X_{-1}, ..., X_n)$$

computes the n first layers of the border of $r_n(d)(x)$, knowing that the last layer is made of x itself, so that $(\mathsf{reflframe}^n(r_{-1}, ..., r_{n-1})(d), x)$ is a full frame, that is of type $\mathsf{frame}^{n+1}(X_{-1}, ..., X_n)$.

Note that the correct typing of $(\mathsf{reflframe}^n(r_{-1}, ..., r_{n-1})(d), x)$, relies, for $X_{-1}, X_0, ..., X_n$ being given, on two familiar coherence conditions:

$$\begin{array}{l} \mathsf{idrestrreflframe}^n(r_{-1}, ..., r_{n-1}) : \Pi d : \mathsf{frame}^n.\, \mathsf{restrframe}^{n,n}_n(\mathsf{reflframe}^n(r_{-1}, ..., r_{n-1})(d)) = d \\ \mathsf{cohrestrreflframe}^n_{p<n}(r_{-1}, ..., r_{n-1}) : \Pi d : \mathsf{frame}^{n,p}. \\ \quad \mathsf{restrframe}^{n,p}_p(\mathsf{reflframe}^{n,p}(r_{-1}, ..., r_{n-1})(d)) = \mathsf{reflframe}^{n-1,p}(r_{-1}, ..., r_{n-2})(\mathsf{restrframe}^{n-1,p}_p(d)) \end{array}$$

where $\mathsf{reflframe}^{n,p}$ generalises $\mathsf{reflframe}^n$ to prefixes of frame^n:

$$\mathsf{reflframe}^{n,p}(r_{-1}, ..., r_{n-1}) : \mathsf{frame}^{n,p}(X_{-1}, ..., X_{n-1}) \to \mathsf{frame}^{n+1,p}(X_{-1}, ..., X_n)$$

The full construction of degeneracies, machine-checked, can be found in Rocq syntax in https://github.com/artagnon/bonak/blob/dgn/theories/ (file νType.v).

Getting rid of the propositional equations used in the recursive argument of [HR25]. The [HR25] construction is highly dependent. For instance, we need the definition of $\mathsf{frame}^{n,p}$ to type $\mathsf{restrframe}^{n,p}$ and we need the definition of $\mathsf{restrframe}^{n,p}$ to define $\mathsf{frame}^{n,p+1}$. In the formalisation associated to [HR25], this is managed by stating the definition of $\mathsf{frame}^{n,p}$ and $\mathsf{restrframe}^{n,p}$ propositionally in the recursive step of the construction. We are close to complete the formalisation of a propositional-equality-free variant obtained by mutually defining the *types* of all $\mathsf{restrframe}^{n,q}$ for $q < p$ together with the *definition body* of $\mathsf{frame}^{n,p}$, itself dependently on the assumption of *definitions* of $\mathsf{restrframe}^{n,q}$ for $q < p$ (and a similar mutual dependency between $\mathsf{restrframe}^{n,p}$ and the types of their coherences $\mathsf{cohrestrframe}^{n,p}$). This makes the construction much more compact.

References

[BCM15] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electr. Notes Theor. Comput. Sci.*, 319:67–82, 2015.

[HR25] Hugo Herbelin and Ramkumar Ramachandra. A parametricity-based formalization of semi-simplicial and semi-cubical sets. To appear in the MSCS special issue on advances in homotopy type theory, 2025.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM Press.

Predicative Stone Duality in Univalent Foundations

Martín H. Escardó and Ayberk Tosun

University of Birmingham
Birmingham, United Kingdom

Summary. Stone duality for spectral locales [8] states that the category of spectral locales and spectral maps is dually equivalent to the category of distributive lattices. In this work, we construct this equivalence in the setting of constructive and predicative univalent foundations (UF), by working with the category of large, locally small, and small-complete spectral locales. This extends the authors' previous work [1] in which a predicatively well-behaved definition of the notion of spectral locale was proposed in the setting of UF. In showing that this notion of spectrality enjoys Stone duality, we formally establish that it captures the intended category of spectral locales.

The results presented here will soon appear in the second named author's PhD thesis [9].

Background on Stone duality. In an impredicative setting, a *spectral locale* [7, p. 63] is defined[1] as a locale in which the compact opens form a base closed under finite meets. A continuous map of spectral locales is called spectral if it reflects compact opens.

A *distributive lattice* (also called *bounded distributive lattice* in the literature) is a lattice with finite meets and finite joins in which the meets distribute over the joins (and vice versa). Given a spectral locale X, the collection of its compact opens (denoted $\mathsf{K}(X)$) forms a distributive lattice as the compact opens are always closed under finite joins, and also closed under finite meets in spectral locales. Conversely, given a distributive lattice L, the collection of ideals of L (denoted $\mathsf{Idl}(L)$) forms a spectral frame. The locale defined by this frame is called the *spectrum of* L and is denoted $\mathsf{spec}(L)$. Stone duality for spectral locales amounts to the fact that these two maps form a categorical equivalence when extended to functors.

Foundations. We work in the context of intensional MLTT extended with univalent features. We assume the existence of Σ and Π types as well as the inductive types of $\mathbf{0}$, $\mathbf{1}$, the natural numbers, and lists. Furthermore, we assume univalence and propositional truncations. Instead of assuming the univalence axiom globally, we assume specific universes in consideration to be univalent. We define the type of families (with index type living in universe $\mathcal{W}$) on a given type A as $\mathsf{Fam}_{\mathcal{W}}(A) :\equiv \Sigma_{(I\,:\,\mathcal{W})} I \to A$. We often use the abbreviation $(x_i)_{i:I}$ for a family $x : I \to A$.

We work explicitly with universes. We denote the ground universe by $\mathcal{U}_0$, the successor of a universe $\mathcal{U}$ by $\mathcal{U}^+$, and the least upper bound of two universes $\mathcal{U}$ and $\mathcal{V}$ by $\mathcal{U} \sqcup \mathcal{V}$.

A type $X : \mathcal{U}$ is called *$\mathcal{V}$-small* if it is equivalent to a specified type in universe $\mathcal{V}$ i.e. $\Sigma_{(Y\,:\,\mathcal{V})} X \simeq Y$, and is called *locally $\mathcal{V}$-small* if the identity type $x =_X y$ is $\mathcal{V}$-small, for every pair of inhabitants $x, y : X$. A universe $\mathcal{U}$ is said to be *univalent* if the canonical map $\mathsf{idtoeqv} : X =_{\mathcal{U}} Y \to X \simeq Y$ is an equivalence, and the *univalence axiom* says that all universes are univalent. The univalence axiom implies that the type expressing that X is $\mathcal{V}$-small is a proposition, for every pair of universes $\mathcal{U}$ and $\mathcal{V}$, and every type $X : \mathcal{U}$. In fact, this can be extended to a logical equivalence [4]: if X being $\mathcal{V}$-small is a proposition for every pair of universes $\mathcal{U}$ and $\mathcal{V}$ and every type $X : \mathcal{U}$, then the univalence axiom holds.

[1] This notion is also called a *coherent locale* in the literature.

Locale theory with explicit universes in UF. We fix a base universe $\mathcal{U}$ and we call $\mathcal{U}$-small types simply *small* in this context. A lattice is called (1) *large* if its carrier set lives in universe $\mathcal{U}^+$ and *small* otherwise, (2) *locally small* if its order has small truth values, and (3) *small-complete* if it has joins of small families. It is well known that complete, small lattices cannot be constructed in a predicative foundational setting without using a form of propositional resizing. This was first shown by Curi [2, 3] for CZF, and then by de Jong and Escardó [5, 4] for predicative UF. Due to this no-go theorem, our investigation of locale theory focuses on the category of large, locally small, and small-complete frames.

Spectral locales in a predicative setting. A family $(B_i)_{i:I}$ of opens in some locale X is said to form a *weak base* for X if, for every open U, there is an unspecified subfamily $(B_{i_j})_{j:J}$ such that $U = \bigvee_{j:J} B_{i_j}$. A locale X is called *spectral* if it satisfies the following four conditions: (SP1) it is compact (i.e. the empty meet is compact), (SP2) the meet of two compact opens is compact, (SP3) the family $\mathsf{K}(X) \hookrightarrow \mathcal{O}(X)$ forms a weak base, and (SP4) the type $\mathsf{K}(X)$ is small.

Compared to the standard definition, the most salient difference here is the stipulation that the type $\mathsf{K}(X)$ be small. The justification for having requirement in the definition is twofold: (1) many fundamental constructions of locale theory (e.g. Heyting implications, open nuclei, right adjoints of frame homomorphisms) do not seem to be available in the absence of a small (weak) base, and (2) a natural desideratum for the definition of spectrality is that it satisfy Stone duality, and it can be shown, as we will explain soon, that *every* locale homeomorphic to the spectrum of a (small) distributive lattice satisfies the above definition. We denote by $\mathbf{Spec}_{\mathcal{U}}$ the category of large, locally small, and small-complete spectral locales over universe $\mathcal{U}$.

Propositionality of being spectral. In UF, it is natural to expect that being spectral is a property and not structure. If we assume the base universe to be univalent, we can show that this is indeed a proposition, since this is what we need to conclude that Condition (SP4) is a proposition. As explained previously, however, the propositionality of being small is logically equivalent to univalence, which is to say that the use of univalence here seems to be essential.

Spectrum construction with explicit universes. A small distributive lattice (over base universe $\mathcal{U}$) is a small set $L : \mathcal{U}$ equipped with operations $\text{-}\wedge\text{-} : L \to L \to L$ and $\text{-}\vee\text{-} : L \to L \to L$, which are associative and commutative, have unit elements, and satisfy the absorption laws i.e. $x \wedge (x \vee y) = x$ and $x \vee (x \wedge y) = x$, for every $x, y : L$. We denote by $\mathbf{DLat}_{\mathcal{U}}$ the category of small distributive lattices over base universe $\mathcal{U}$. By a *small ideal* of L, we mean a $\mathcal{U}$-valued subset $S : L \to \Omega_{\mathcal{U}}$ that is inhabited, downward closed, and closed under binary joins. We show that the small ideals of a lattice L form a large, locally small, and small-complete locale satisfying the above definition of spectrality. We then extend this to a functor $\mathsf{spec} : \mathbf{DLat}_{\mathcal{U}} \to \mathbf{Spec}_{\mathcal{U}}$.

Predicative Stone duality. We show that the type $\mathsf{K}(X)$ is a small distributive lattice, for every large, locally small, and small-complete spectral locale X. This exploits Condition (SP4), stipulating the smallness of $\mathsf{K}(X)$. We then extend this to a functor $\mathsf{K} : \mathbf{Spec}_{\mathcal{U}} \to \mathbf{DLat}_{\mathcal{U}}$, and show that it forms a categorical equivalence when paired with $\mathsf{spec} : \mathbf{DLat}_{\mathcal{U}} \to \mathbf{Spec}_{\mathcal{U}}$.

Formalization. Most of the work we present has been formalized using the Agda proof assistant, as part of the TypeTopology library [6]. The object part of the categorical equivalence has been completely formalized, and the extension of this to the full categorical equivalence is work in progress at the time of writing.

References

[1] Igor Arrieta, Martín H. Escardó, and Ayberk Tosun. The patch topology in univalent foundations, 2024. arXiv: 2402.03134 [cs.LO].

[2] Giovanni Curi. On some peculiar aspects of the constructive theory of point-free spaces. *Mathematical Logic Quarterly*, 56(4):375–387, 2010.

[3] Giovanni Curi. On the existence of Stone–Čech compactification. *The Journal of Symbolic Logic*, 75(4):1137–1146, 2010.

[4] Tom de Jong and Martín H. Escardó. On small types in univalent foundations. *Logical Methods in Computer Science*, Volume 19, Issue 2.

[5] Tom de Jong and Martín H. Escardó. Predicative aspects of order theory in univalent foundations. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, volume 195 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[6] Martín H. Escardó and contributors. TypeTopology. Agda library available at https://github.com/martinescardo/TypeTopology.

[7] Peter T. Johstone. *Stone Spaces*. Cambridge Studies in Advanced Mathematics.

[8] Marshall H. Stone. Topological representation of distributive lattices and Brouwerian logics. *Časopis pro pěstování matematiky a fysiky*, 67:1–25, 1937.

[9] Ayberk Tosun. *Constructive and Predicative Locale Theory in Univalent Foundations*. PhD thesis, University of Birmingham, 2025. Accepted, to be published.

Cocompleteness in simplicial homotopy type theory

Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz

1 Introduction

Simplicial type theory (STT) was introduced by Riehl and Shulman [8] to give a *synthetic* account of ∞-category theory[1] based on homotopy type theory. Specifically, STT extends HoTT with a handful of axioms to tune the theory to a particular collection of models (those based on the ∞-topoi $\mathcal{E}^{\Delta^{\mathrm{op}}}$) into which the ∞-category of small (internal) ∞-categories fully-faithfully embeds [7]. The most important axiom postulates a type $\mathbb{I}$ which behaves like a *directed interval*:

Axiom 1. *We assume an h-set $\mathbb{I} : \mathcal{U}$ such that $\mathbb{I}$ is a totally-ordered bounded distributive lattice.*

We may then use $\mathbb{I}$ to associate to any $x, y : X$ the type of *synthetic morphisms* $\hom(x, y)$:

$$\hom(x, y) = \textstyle\sum_{f:\mathbb{I}\to X} f(0) = x \times f(1) = y$$

Not every type admits a composition operation, so we isolate the data of a composition of two morphisms using $\Delta^2 = \sum_{i,j:\mathbb{I}} i \geq j$. In particular, $\alpha : \Delta^2 \to X$ intuitively classifies a pair of composable morphisms in X—$\alpha(1, -)$ and $\alpha(-, 0)$—together with their composite—$\lambda i.\, \alpha(i, i)$.

Definition 1.1. A pre-category X is a type where the restriction $X^{\Delta^2} \to X^{\mathbb{I}} \times_X X^{\mathbb{I}}$ is an equivalence. An inverse to this restriction is denoted $\circ$ and composes morphisms in X.

While a pre-category has composition operation, to correctly model ∞-categories we further restrain X such that $x =_X y$ coincides with synthetic isomorphisms $x \cong y$:

Definition 1.2. A pre-category X is a category if $\mathsf{isEquiv}\big(\lambda x.(x, x, \mathsf{id}) : X \to \sum_{x,y} x \cong y\big)$.

Definition 1.3. A groupoid is a category where all morphisms are isomorphisms.

On top of this handful of basic definitions, Riehl and Shulman [8] developed the theory of functors, natural transformations, and adjoints within STT and since then authors [1–3, 5, 6, 10, 11] have worked to reproduce or extend ∞-category theory within STT. In particular, the recent work by Gratzer et al. [5, 6] extends STT with a collection of *modalities* [4] to construct the category of groupoids $\mathcal{S}$ along with the theory of presheaf categories.

We extend this line of research on modal STT with a series of results in showing that cocompleteness may be reduced to the existence of various simpler colimits. We are then able to provide an entirely synthetic account of (generalized) homology and cohomology theories.

Notation 1.4. While the details of the modal STT are not relevant, we note that it generalizes Shulman [9] and, in particular, includes the ability to quantify over only the objects of categories i.e., to treat such elements non-functorially. These non-functorial variables are annotated by $\flat$.

2 Colimits and cocompleteness

Our focus is to analyze conditions equivalent to the following:

Definition 2.1. A category C is *cocomplete* if $C \to C^I$ is a right adjoint for all $I :_\flat \mathcal{U}_0$.

[1] By ∞-category theory we specifically mean $(\infty, 1)$-category theory.

Our main results offer simpler alternative conditions for a category to be cocomplete.

Theorem 2.2. *A category C is cocomplete if any of the following hold:*

1. *C has pushouts as well as colimits indexed by groupoids* and *crisp excluded middle holds* (*for every h-proposition $\phi :_\flat \mathcal{U}$ either ϕ or $\neg\phi$ holds*).
2. *C has finite coproducts and all sifted colimits*
3. *C has all finite colimits and all filtered colimits.*

The latter two conditions are both necessary and sufficient.

In the above, filtered and sifted colimits are defined using the notion of cofinal maps as introduced in STT by Gratzer, Weinberger, and Buchholtz [6] and closely follow the standard definitions from ∞-category theory: A category C is sifted if $C \to C^n$ is right cofinal for all $n : \mathsf{Nat}$ and filtered if $C \to C^K$ is right cofinal for all finite complexes K. In fact, in *ibid.* it is shown that the category of $\mathcal{S}$-valued presheaves on C is the free cocompletion.

3 Spectra and (co)homology

We apply the above results to the category of *spectra* Sp. If (∞-)groupoids replace the category of sets in ∞-category theory, spectra take the place of abelian groups. We are now able to show that Sp is stable (Lemma 3.3) and use this to construct homology theories satisfying the Eilenberg–Steenrod axioms.

Definition 3.1. Sp is given by the limit (computed as in Book HoTT) $\varprojlim(\mathcal{S}_* \xleftarrow{\Omega} \mathcal{S}_* \xleftarrow{\Omega} \dots)$.

Lemma 3.2. Sp *has all filtered colimits and all limits.*

Lemma 3.3. Sp *is finitely (co)complete,* $\mathbf{0}_{\mathsf{Sp}} \cong \mathbf{1}_{\mathsf{Sp}}$*, and pushouts and pullbacks coincide.*

Corollary 3.4. Sp *is cocomplete.*

Proof. Apply Theorem 2.2 to Lemmas 3.2 and 3.3. □

A fundamental example of a spectrum is $H\mathbb{Z}$, the Eilenberg–MacLane spectrum given by the sequence of Eilenberg–MacLane spaces $(K(\mathbb{Z},0), K(\mathbb{Z},1), \dots)$. By Gratzer, Weinberger, and Buchholtz [6], there is an equivalence between cocontinuous maps $\mathcal{S} \to \mathsf{Sp}$ and elements of spectra and we write $H : \mathcal{S} \to \mathsf{Sp}$ for the cocontinuous functor induced by $H\mathbb{Z}$. We further write $\pi_i : \mathsf{Sp} \to \mathsf{Ab}$ for the functor sending X to $\pi_0(\mathsf{proj}_i(X))$.

Lemma 3.5. *If $X \to Y$ and $Z = Y \sqcup_X \mathbf{1}$ then there is a long exact sequence of abelian groups:*

$$\pi_i X \longrightarrow \pi_i Y \longrightarrow \pi_i Z \longrightarrow \pi_{i-1} X \longrightarrow \dots$$

Theorem 3.6. *The functors $\pi_i \circ H : \mathcal{S} \to \mathsf{Ab}$ satisfies the Eilenberg–Steenrod axioms.*

Proof. Of the Eilenberg–Steenrod axioms (homotopy invariance, excision, dimension, and additivity), homotopy invariance and dimension follow more-or-less by definition. Excision states that cofibers in $\mathcal{S}$ are sent to long exact sequences. This is a corollary of the cocontinuity of H alongside Lemma 3.5. The additivity axiom states that that coproducts are preserved by H_i. For finite coproducts, Lemma 3.3 implies that finite coproducts agree with finite products and $\pi_i : \mathsf{Sp} \to \mathsf{Ab}$ sends finite products to direct sums by calculations. For the general case, one decomposes $\coprod_{i:I} X_i$ for a discrete set I into the filtered colimit of finite colimits $\varinjlim_{(n,f):\sum_{n:\mathbb{N}} I^n} \coprod_{k \le n} X_{f(i)}$ to reduce to the finite case. □

References

[1] César Bardomiano Martínez. *Exponentiable functors between synthetic ∞-categories*. 2024. arXiv: 2407.18072.

[2] César Bardomiano Martínez. *Limits and colimits of synthetic ∞-categories*. 2022. arXiv: 2202.12386.

[3] Ulrik Buchholtz and Jonathan Weinberger. "Synthetic fibered $(\infty, 1)$-category theory". In: *Higher Structures* 7 (1 2023), pp. 74–165. DOI: 10.21136/HS.2023.04.

[4] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. "Multimodal Dependent Type Theory". In: *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021). DOI: 10.46298/lmcs-17(3:11)2021.

[5] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. *Directed univalence in simplicial homotopy type theory*. 2024. arXiv: 2407.09146 [cs.LO]. URL: https://arxiv.org/abs/2407.09146.

[6] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. *The Yoneda embedding in simplicial type theory*. 2025. arXiv: 2501.13229 [cs.LO]. URL: https://arxiv.org/abs/2501.13229.

[7] Charles Rezk. "A model for the homotopy theory of homotopy theory". In: *Trans. Amer. Math. Soc.* 353.3 (2001), pp. 973–1007. DOI: 10.1090/S0002-9947-00-02653-2.

[8] Emily Riehl and Michael Shulman. "A type theory for synthetic ∞-categories". In: *Higher Structures* 1 (1 2017), pp. 147–224. DOI: 10.21136/HS.2017.06.

[9] Michael Shulman. "Brouwer's fixed-point theorem in real-cohesive homotopy type theory". In: *Mathematical Structures in Computer Science* 28.6 (2018), pp. 856–941. DOI: 10.1017/S0960129517000147. URL: https://doi.org/10.1017/S0960129517000147.

[10] Jonathan Weinberger. "A Synthetic Perspective on $(\infty, 1)$-Category Theory: Fibrational and Semantic Aspects". PhD thesis. Technische Universität Darmstadt, 2022. DOI: 10.26083/tuprints-00020716.

[11] Jonathan Weinberger. "Internal sums for synthetic fibered $(\infty, 1)$-categories". In: *Journal of Pure and Applied Algebra* 228.9 (Sept. 2024), p. 107659. ISSN: 0022-4049. DOI: 10.1016/j.jpaa.2024.107659. URL: http://dx.doi.org/10.1016/j.jpaa.2024.107659.

Yet another homotopy group, yet another Brunerie number

Tom Jack
Independent researcher, USA
pi3js2@proton.me

Axel Ljungström
Stockholm University, Sweden
axel.ljungstrom@math.su.se

Homotopy type theory (HoTT) has been proposed as a foundation of synthetic homotopy theory and its usefulness was witnessed early on by e.g. Brunerie's 2016 proof that $\pi_4(\mathbb{S}^3)$ – the fourth homotopy group of the 3-sphere – is isomorphic to $\mathbb{Z}/2\mathbb{Z}$ [Bru16]. As Brunerie's proof appeared only 78 years after Pontrjagin's original proof [Pon38], it is about time (if we wish to keep up with the classical homotopy theorists) we start thinking about the next homotopy group in line: $\pi_5(\mathbb{S}^3)$. In this note, we present a proof that $\pi_5(\mathbb{S}^3) \cong \mathbb{Z}/n\mathbb{Z}$ for some (constructively defined) $n \in \{1, 2\}$. We have not yet been able to carry out a (terminating) normalisation of this number in a constructive proof assistant such as `Cubical Agda`, but we are hopeful – Pontrjagin and Whitehead only showed that $\pi_5(\mathbb{S}^3) \cong \mathbb{Z}/n\mathbb{Z}$ in 1950[1] [Pon50; Whi50], so we can let `Agda` think for another couple of years before we hit the 78-year mark.

Concretely, our first key contribution – Theorem 1 – is a HoTT version of a theorem by Gray [Gra73] which relates cofibres of certain so-called *Whitehead products* to the fibre of the *pinch map*. Using that such cofibres are well known to be related to the homotopy groups of spheres, we are able to use Theorem 1 to derive our second key contribution – Theorem 2 – which is a characterisation of $\pi_5(\mathbb{S}^3)$. While we appreciate that Theorem 1 is a somewhat niche result, we hope that Theorem 2 will be of interest to a broader audience of type theorists as it presents a so-called 'Brunerie number'. That is, it presents a constructively defined $n \in \{1, 2\}$ whose normalisation in a constructive proof assistant like `Cubical Agda` would yield a proof of a non-trivial theorem: in this case, of the fact that $\pi_5(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$. We will discuss our struggles with actually normalising/computing this number. While the formalisation of the proof that $\pi_5(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$ is in its early stages, the construction of our new Brunerie number is independent and can be defined in `Cubical Agda` with little effort using the formalisation of $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$ due to Ljungström and Mörtberg [LM23]. We remark that we do also have a pen-and-paper proof of the fact that our Brunerie number is 2. Nevertheless, we are still interested in producing a fully computer-assisted proof.

Preliminaries/notation We will not go into any detailed proofs and will not require too much knowledge of classical homotopy theory. We do, however, rely on some basics of HoTT which we briefly rush through here. The reader comfortable with HoTT can skim this section.

Pointed types and maps: a pointed type is a pair $(A, \star_A)$ where $\star_A : A$. We will simply write A and leave $\star_A$ implicit. We use the same convention for pointed maps and write $f : A \to_\star B$ for a pointed map from A to B and leave the proof $\star_f : f(\star_A) = \star_B$ implicit.

Loop spaces: given a pointed type A, we let $\Omega(A) := (\star_A = \star_A)$ denote its loop space. Given a pointed map $f : A \to_\star B$, we write $\Omega\, f : \Omega(A) \to_\star \Omega(B)$ for the functorial action of Ω.

Pushouts: given a span $B \xleftarrow{f} A \xrightarrow{g} D$, we can form its pushout. This is the higher inductive type (HIT) $P_{f,g}$ with constructors $\mathsf{inl} : B \to P_{f,g}$ $\quad \mathsf{inr} : D \to P_{f,g}$ $\quad \mathsf{push} : (a : A) \to f(a) = g(a).$ Pushouts are always taken to be pointed by $\mathsf{inl}(\star_B)$ when B is pointed. Important instances are:

Name	Notation	Pushout of span	Comments
Join	$A * B$	$A \leftarrow A \times B \rightarrow B$	
Cofibre (of f)	C_f	$\mathbb{1} \leftarrow A \xrightarrow{f} B$	
Wedge sum	$A \vee B$	$A \xleftarrow{x \mapsto \star_A} \mathbb{1} \xrightarrow{x \mapsto \star_B} B$	A and B are pointed types
Suspension	ΣA	$\mathbb{1} \leftarrow A \rightarrow \mathbb{1}$	When A is pointed, there is a 'suspension function' $\sigma_A : A \to \Omega(\Sigma A)$ given by $\sigma_A(x) := \mathsf{push}(x) \cdot \mathsf{push}(\star_A)^{-1}$

Recall also that the n-sphere can be defined in terms of suspension by defining it to be the $(n+1)$-fold suspension of the empty type, i.e. $\mathbb{S}^n := \Sigma^{n+1}\bot$.

[1] The original proofs concerned $\pi_6(\mathbb{S}^4)$ but this group is isomorphic to the one in question by the quaternionic Hopf fibration [BR18].

Homotopy groups: given a pointed type A, we define its nth homotopy group by $\pi_n(A) := \| \mathbb{S}^n \to_\star A \|_0$.[2] Here, $\| - \|_0$ denotes *set truncation*, i.e. the operation which takes a type and forces it to satisfy UIP. The type $\pi_n(A)$ has a natural group structure when $n \geq 1$ which is abelian when $n \geq 2$. Given a pointed map $f : A \to_\star B$, there is an induced homomorphism $f_* : \pi_n(A) \to \pi_n(B)$. This yields a long exact sequence $\cdots \to \pi_{n+1}(B) \to \pi_n(\mathsf{fib}_f) \xrightarrow{\mathsf{fst}_*} \pi_n(A) \xrightarrow{f_*} \pi_n(B) \to \pi_{n-1}(\mathsf{fib}_f) \to \dots$ where fib_f is short for $\mathsf{fib}_f(\star_B)$, the fibre of f over $\star_B$, i.e. $(a : A) \times (f(a) = \star_B)$.

Connectedness: a type A is n-connected if its n-truncation, $\| A \|_n$, is contractible. We say that a map $f : A \to B$ is n-connected if all its fibres are n-connected. The key thing we will need to know about connectedness is that if $f : A \to B$ is an n-connected and pointed map, then the induced map $f_* : \pi_m(A) \to \pi_m(B)$ is an isomorphism when $m \leq n$ and surjective when $m = n + 1$.

The pinch map: an important map for us will be the so-called pinch map. Given a function $f : A \to B$, we define $\mathsf{pinch}_f : C_f \to \Sigma A$ by

$$\mathsf{pinch}_f(\mathsf{inl}(\star_{\mathbb{1}})) := \mathsf{inl}(\star_{\mathbb{1}}) \quad \mathsf{pinch}_f(\mathsf{inl}(b)) := \mathsf{inr}(\star_{\mathbb{1}}) \quad \mathsf{ap}_{\mathsf{pinch}_f}(\mathsf{push}(a)) := \mathsf{push}(a).$$

The main results In order to state our main results, we will need a key construction from homotopy theory called *Whitehead products* [Ark62]. These turn homotopy groups into a graded Lie superalgebra and provide us with principled ways of constructing elements of homotopy groups. All you need to know for this presentation, however, is that (just like the original Brunerie number [Bru16]), the Brunerie number we present here will be defined in terms of a certain Whitehead product. The most bare-bones definition of the Whitehead product is called the *generalised Whitehead product*: given two functions $f : \Sigma A \to D$, $g : \Sigma B \to D$, we define their (generalised) Whitehead product to be the function $[f, g] : A * B \to D$ defined by

$$[f,g](\mathsf{inl}(a)) := \star_D \quad [f,g](\mathsf{inr}(b)) := \star_D \quad \mathsf{ap}_{[f,g]}(\mathsf{push}(a)) := (\Omega\, g)(\sigma_B(b)) \cdot (\Omega\, f)(\sigma_A(a))$$

This definition follows those of Brunerie and Ljungström & Mörtberg [Bru16; LM24].

It turns out that some homotopy groups of spheres correspond to homotopy groups of cofibres of certain Whitehead products. Indeed, Brunerie showed that $\pi_{k+1}(\mathbb{S}^{n+1}) \cong \pi_k(C_{[\mathsf{id}_{\mathbb{S}^n}, \mathsf{id}_{\mathbb{S}^n}]})$ for $k \leq 3n - 2$.[3] This allowed him to define his (in)famous Brunerie number and, later, characterise $\pi_4(\mathbb{S}^3)$. Here, we are interested in the next homotopy group, $\pi_5(\mathbb{S}^3)$ and, as it happens, this group too can be understood in terms of the cofibre of a Whitehead product. To get there, let us state our first main theorem – it is a HoTT counterpart of a classical theorem by Gray [Gra73] and relates the (homotopy groups of) cofibres of Whitehead products of the form $[\mathsf{id}_{\Sigma B}, f]$ to fibres of the pinch map.

Theorem 1. *Let A be an $(a-1)$-connected pointed type and B be any pointed type. Let $f : \Sigma A \to_\star \Sigma B$. In this case, there is a $2a$-connected map $\gamma : C_{[\mathsf{id}_{\Sigma B}, f]} \to \mathsf{fib}_{\mathsf{pinch}_f}$ where, recall, $\mathsf{pinch}_f : C_f \to \Sigma^2 A$.*

Assume further that B in Theorem 1 is $(b-1)$-connected. In this case, we get that $\pi_n(\mathsf{fib}_{\mathsf{pinch}_f}) \cong \pi_n(\Sigma B)$ when $n \leq a + b$ for elementary connectedness reasons. Combining this information with Theorem 1 and the long exact sequence associated with the fibration sequence $\mathsf{fib}_{\mathsf{pinch}_f} \to C_f \to \Sigma^2 A$, we obtain a new sequence of the form

$$\cdots \to \pi_{n+1}(\Sigma^2 A) \to \pi_n(F_n) \to \pi_n(C_f) \to \pi_n(\Sigma^2 A) \to \pi_{n-1}(F_{n-1}) \to \dots \quad \text{where}$$
$$F_{n \leq a+b} := \Sigma B \qquad F_{a+b<n\leq 2a} := C_{[\mathsf{id}_{\Sigma B}, f]} \qquad F_{2a<n} := \mathsf{fib}_{\mathsf{pinch}_f}$$

Let us instantiate this sequence with $A := \mathbb{S}^2$, $B := \mathbb{S}^1$, $f := [\mathsf{id}_{\mathbb{S}^2}, \mathsf{id}_{\mathbb{S}^2}]$, $a = 2$ and $b = 1$. Note that this is well-typed because the domain of f is $\mathbb{S}^1 * \mathbb{S}^1$ which is equivalent to $\mathbb{S}^3$. The conflation of these spaces will be used without comment from now on. We get

$$\pi_5(\mathbb{S}^4) \to \pi_4(C_{[\mathsf{id}_{\mathbb{S}^2}, [\mathsf{id}_{\mathbb{S}^2}, \mathsf{id}_{\mathbb{S}^2}]]}) \to \pi_4(C_{[\mathsf{id}_{\mathbb{S}^2}, \mathsf{id}_{\mathbb{S}^2}]}) \to \pi_4(\mathbb{S}^4) \to \pi_3(\mathbb{S}^2) \to \pi_3(C_{[\mathsf{id}_{\mathbb{S}^2}, \mathsf{id}_{\mathbb{S}^2}]}).$$

All of these groups have alternative descriptions.

- The first group, $\pi_5(\mathbb{S}^4)$, is isomorphic to $\pi_4(\mathbb{S}^3)$ by stability [UF13, Corollary 8.6.15], which we know is further isomorphic to $\mathbb{Z}/2\mathbb{Z}$.

[2] We could equivalently have set $\pi_n(A) := \| \Omega^n(A) \|_0$. While this definition makes the group structure on $\pi_n(A)$ very clear (it is simply path composition), it makes some other constructions (most importantly for us, Whitehead products) somewhat more roundabout.

[3] Concretely, Brunerie shows in the proof of Proposition 3.4.4 that $J_2(\mathbb{S}^n) \simeq C_{[\mathsf{id}_{\mathbb{S}^n}, \mathsf{id}_{\mathbb{S}^n}]}$, where the former type denotes the second type in the *James construction* on $\mathbb{S}^n$ – a type which, by Brunerie's Proposition 3.2.1, has π_k isomorphic to $\pi_{k+1}(\mathbb{S}^{n+1})$ for $k \leq 3n - 2$.

- For the second group, we have that $[\mathsf{id}_{\mathbb{S}^2}, [\mathsf{id}_{\mathbb{S}^2}, \mathsf{id}_{\mathbb{S}^2}]] = [\mathsf{id}_{\mathbb{S}^2}, 2\eta] = 2[\mathsf{id}_{\mathbb{S}^2}, \eta] = 0$ where the first equality is (in essence) the original Brunerie number, the second follows from bilinearity of Whitehead products [Lju25] and the third follows from 2-torsion. Thus, $C_{[\mathsf{id}_{\mathbb{S}^2}, [\mathsf{id}_{\mathbb{S}^2}, \mathsf{id}_{\mathbb{S}^2}]]}$ is the cofibre of the constant map $\mathbb{S}^4 \to_\star \mathbb{S}^2$ and is, as such, equivalent to $\mathbb{S}^5 \vee \mathbb{S}^2$. Furthermore, we have $\pi_4(\mathbb{S}^5 \vee \mathbb{S}^2) \cong \pi_4(\mathbb{S}^5) \times \pi_4(\mathbb{S}^2) \cong \pi_4(\mathbb{S}^2) \cong \pi_4(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$, where the second to last isomorphism comes from the Hopf map [Bru16, Proposition 2.6.8].
- The third and sixth groups, $\pi_n(C_{[\mathsf{id}_{\mathbb{S}^2}, \mathsf{id}_{\mathbb{S}^2}]})$ for $n \in \{3, 4\}$, are isomorphic to $\pi_{n+1}(\mathbb{S}^3)$ by (as mentioned briefly earlier) Brunerie's work [Bru16, Section 3.4].
- The fourth and fifth groups, $\pi_4(\mathbb{S}^4)$ and $\pi_3(\mathbb{S}^2)$ respectively, are well known to be isomorphic to the integers [UF13, Theorem 8.6.17, Corollary 8.6.19].

With this information, we can rewrite this instance of the sequence as follows.

$$\mathbb{Z}/2\mathbb{Z} \xrightarrow{d} \mathbb{Z}/2\mathbb{Z} \to \pi_5(\mathbb{S}^3) \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}/2\mathbb{Z}$$

It follows completely abstractly, i.e. without knowing any details about either of the maps in the above sequence, that the second map must be surjective, and thus we have constructed our Brunerie number:

Theorem 2. $\pi_5(\mathbb{S}^3) \cong \mathbb{Z}/(2 - d(1))\mathbb{Z}$.

The situation we find ourselves in now should seem awfully familiar to anyone familiar with Brunerie's thesis... We need to compute $d(1) : \mathbb{Z}/2\mathbb{Z}$. By unfolding its definition, we see that this number can be understood as the result of applying the isomorphism $\phi : \pi_4(\mathbb{S}^2) \cong \mathbb{Z}/2\mathbb{Z}$ (in HoTT, due to Brunerie [Bru16] and formalised by Ljungström & Mörtberg [LM23]) to the composite map $\mathbb{S}^4 \xrightarrow{\Sigma\eta} \mathbb{S}^3 \xrightarrow{[\mathsf{id}_{\mathbb{S}^2}, \mathsf{id}_{\mathbb{S}^2}]} \mathbb{S}^2$ (viewed as an element of $\pi_4(\mathbb{S}^3)$) where $\eta : \pi_3(\mathbb{S}^2)$ is the generator. This composite map is equal to $\mathbb{S}^4 \xrightarrow{\Sigma\eta} \mathbb{S}^3 \xrightarrow{2\eta} \mathbb{S}^2$. Although the easiest approach would be to simply compute $d(1) := \phi(2\eta \circ \Sigma\eta)$ in `Cubical Agda`, we have not had much success.[4] It is, however, relatively easy to prove that $d(1) = 0$ by showing that $2\eta \circ \Sigma\eta = 0$ by hand. This is a consequence of the fact that the precomposition map $(-) \circ \Sigma\eta$ defines a group homomorphism $\pi_3(\mathbb{S}^2) \to \pi_4(\mathbb{S}^2)$ and therefore must vanish on 2η due to 2-torsion in $\pi_4(\mathbb{S}^2)$. Hence, we may conclude that $\pi_5(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$.

References

[Ark62] M. Arkowitz. "The generalized Whitehead product." In: *Pacific Journal of Mathematics* 12.1 (1962), pp. 7–23. DOI: 10.2140/pjm.1962.12.7.

[Bru16] G. Brunerie. "On the homotopy groups of spheres in homotopy type theory". PhD thesis. Université Nice Sophia Antipolis, 2016. arXiv: 1606.05916.

[BR18] U. Buchholtz and E. Rijke. "The Cayley-Dickson Construction in Homotopy Type Theory". In: *Higher Structures* 2.1 (2018), pp. 30–41. DOI: 10.21136/HS.2018.02.

[Cub24] The Agda Community. *Cubical Agda Library*. Version 0.7. Feb. 2024. URL: https://github.com/agda/cubical.

[Gra73] B. Gray. "On the Homotopy Groups of Mapping Cones". In: *Proceedings of the London Mathematical Society* s3-26.3 (Apr. 1973), pp. 497–520. ISSN: 0024-6115. DOI: 10.1112/plms/s3-26.3.497.

[LM23] A. Ljungstrom and A. Mortberg. "Formalizing $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$ and Computing a Brunerie Number in Cubical Agda". In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2023, pp. 1–13. DOI: 10.1109/LICS56636.2023.10175833.

[Lju25] A. Ljungström. *Some properties of Whitehead products*. Extended abstract at *Workshop on Homotopy Type Theory/Univalent Foundations (HoTT/UF 2025)*. 2025. URL: https://hott-uf.github.io/2025/abstracts/HoTTUF_2025_paper_23.pdf.

[LM24] A. Ljungström and A. Mörtberg. *Formalising and Computing the Fourth Homotopy Group of the 3-Sphere in Cubical Agda*. 2024. arXiv: 2302.00151.

[Pon38] L. Pontrjagin. "Classification of continuous maps of a complex into a sphere, Communication I". In: *Doklady Akademii Nauk SSSR* 19.3 (1938), pp. 147–149.

[4] The number $d(1)$ is easy to implement since the only components needed are the Hopf map and the isomorphism $\pi_4(\mathbb{S}^2) \cong \mathbb{Z}/2\mathbb{Z}$. These constructions were already available in the `Cubical Agda` library [Cub24] at the start of this project.

[Pon50] L. Pontrjagin. "Homotopy classification of mappings of an $(n+2)$-dimensional sphere on an n-dimensional one". In: *Doklady Akademii Nauk SSSR* 70.6 (1950), pp. 957–959.

[UF13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: Self-published, 2013. URL: https://homotopytypetheory.org/book/.

[Whi50] G. W. Whitehead. "The $(n+2)^{\text{nd}}$ Homotopy Group of the n-Sphere". In: *Annals of Mathematics* 52.2 (1950), pp. 245–247.

Rezk Completions For (Elementary) Topoi

Kobe Wullaert[1] and Niels van der Weide[2]

[1] Delft University of Technology, The Netherlands
[2] Radboud University, The Netherlands

We continue the development of univalent category theory in univalent foundations. Recall that the Rezk completion [2] provides a universal solution to the problem of constructing a univalent category from an arbitrary category. In this work, we lift the Rezk completion from categories to elementary topoi. The results presented below are formalized in (Coq-)UniMath [6][1].

1 Introduction

Internal to univalent foundations, the "well-behaved categories" are those satisfying a certain coherence condition: *univalence*. For every category, one always has a canonical function $idtoiso_{x,y}$ from the identity type $x = y$, between two objects x and y, to the type $x \cong y$ of isomorphisms between them. A **univalent category** is a category for which $idtoiso_{x,y}$ is an equivalence of types, for every x and y. The univalence requirement for categories can be motivated by to the variety of examples which satisfy this requirement, and by to the intended semantics of categories in e.g., Voevodsky's simplicial set model of HoTT/UF.

Univalent categories are particulary well-behaved. First, notions that are classically unique up to isomorphism, such as limits, become unique up to identity when working with univalent categories. Second, isomorphisms of categories coincide with equivalences. Hence, structures are automatically invariant under equivalences.

Even though many occuring categories are in fact univalent, certain constructions on categories can fail to produce univalent categories. For example, in categorical logic, the construction of a topos from a tripos (a.k.a., a second-order hyperdoctrine), often produces a non-univalent topos [3, 4]. Nonetheless, for every category $\mathcal{C}$, we can construct a univalent category $\mathsf{RC}(\mathcal{C})$ and a weak equivalence $\eta_{\mathcal{C}}$ from $\mathcal{C}$ to $\mathsf{RC}(\mathcal{C})$ [2]. That is, there is a functor $\eta_{\mathcal{C}} : \mathcal{C} \to \mathsf{RC}(\mathcal{C})$ which is fully faithful and essentially surjective on objects; the latter condition means that for every $y : \mathsf{RC}(\mathcal{C})$, there *merely* exists some $x : \mathcal{C}$ and an isomorphism between $F(x)$ and y. Furthermore, $(\mathsf{RC}(\mathcal{C}), \eta_{\mathcal{C}})$ is universal in the sense that every functor from $\mathcal{C}$ to a univalent category $\mathcal{E}$ can be uniquely extended to a functor of type $\mathsf{RC}(\mathcal{C}) \to \mathcal{E}$. Given a category $\mathcal{C}$, the pair $(\mathsf{RC}(\mathcal{C}), \eta_{\mathcal{C}})$ is referred to as **the Rezk completion**. Motivated by the tripos-to-topos construction, we lift the Rezk completion, from categories, to categories with additional structure; in particular, to elementary topoi.

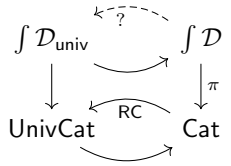
To prove that the Rezk completion lifts to topoi, it suffices that each of the ingredients defining topoi suitably transports along those weak equivalences given by the Rezk completion. This incremental proof strategy is made precise through the language of displayed (bi)categories [1].

[1] https://github.com/UniMath/UniMath/blob/master/UniMath/Bicategories/RezkCompletions/DisplayedRezkCompletions.v,https://github.com/UniMath/UniMath/tree/master/UniMath/Bicategories/RezkCompletions/StructuredCats, https://github.com/UniMath/UniMath/pull/2035

2 Framework

The universal property characterizing the Rezk completion states that every functor into a univalent category factors uniquely through the Rezk completion. The universal property implies that the inclusion of (the bicategory of) univalent categories $\mathsf{UnivCat}$, into all categories Cat, has a left biadjoint RC, whose action on objects is given by the Rezk completion. Hence, to lift the Rezk completion to categories with additional structure, we lift the left biadjoint to the bicategory whose objects are those *structured categories* and whose morphisms are *structure preserving functors*. For simplicity, we assume that the 2-cells are *all* natural transformations; an assumption shared by each of the structures characterizing elementary topoi.

Those bicategories of structured categories inherit much of the structure of the bicategory of categories. Hence, to reuse the underlying structure, we rely on the theory of displayed bicategories. To this end, let $\mathcal{D}$ be a displayed bicategory over Cat encoding some structure for categories, whose total bicategory is denoted $\int \mathcal{D}$. The restriction of $\mathcal{D}$ to $\mathsf{UnivCat}$ is denoted $\mathcal{D}_{\mathsf{univ}}$. Then, the lifting corresponds to the construction of a left biadjoint as depicted in the following diagram:



To construct the left biadjoint, we use that the unit η corresponds pointwise with weak equivalences into univalent categories. The precise properties that are to be verified are summarized in the following lemma, where local contractibility of $\mathcal{D}$ means that every type of displayed 2-cells is contractible.

Lemma 1. Let $\mathcal{D}$ be a locally contractible displayed bicategory over Cat such that

1. for every weak equivalence $G : \mathcal{C}_1 \to \mathcal{C}_2$, whose codomain is univalent, and $x : \mathcal{D}(\mathcal{C}_1)$, there is a given $\hat{x} : \mathcal{D}(\mathcal{C}_2)$ and a displayed morphism $x \to_G \hat{x}$;
2. for every univalent category $\mathcal{C}_3$, natural isomorphism $\alpha : (G \cdot H) \Rightarrow F$, terms $x_i : \mathcal{D}(\mathcal{C}_i)$ $(i = 1, 2, 3)$, and $f : x_1 \to_F x_2$, there is a given $\hat{f} : x_2 \to_H x_3$.

Then, the pseudofunctor $\mathsf{RC} : \mathsf{Cat} \to \mathsf{UnivCat}$ lifts to a biadjoint for $\int \mathcal{D}_{\mathsf{univ}} \hookrightarrow \int \mathcal{D}$.

First, observe that we do not use any particular construction of the Rezk completion. The first condition in Lemma 1 states that every structure transports to the codomain of such a weak equivalence and that the functor preserves the structure. In particular, the first condition also holds for $\mathcal{C}_2 := \mathsf{RC}(\mathcal{C}_1)$ and $G := \eta_{\mathcal{C}_1}$. Hence, the Rezk completion inherits the structure given by $\mathcal{D}$ and $\eta_{\mathcal{C}_1}$ preserves the inherited structure. The second condition expresses that $(\mathsf{RC}(\mathcal{C}_1), \eta_{\mathcal{C}_1})$ not only lives in $\int \mathcal{D}_{\mathsf{univ}}$, but is universal w.r.t., those univalent categories which have the structure. Furthermore, the functoriality of $\int \mathcal{D} \to \int \mathcal{D}_{\mathsf{univ}}$ follows from the second condition. Observe that it suffices to only provide the existence part of the universality condition since the uniqueness follows necessarily from the local contractibility condition.

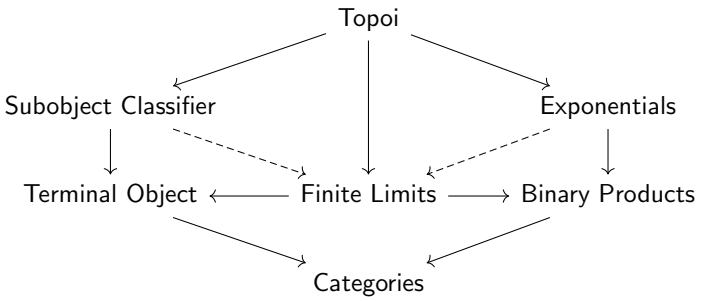
The assumptions of Lemma 1 are illustrated in the following example:

Example 2. Let $\mathcal{D}$ be the Cat-displayed bicategory whose displayed objects are binary products, and whose displayed morphisms witnesses the preservation of those binary products. Then, condition 1 instantiates to the following statement: For G as above, and x a choice of binary

products on $\mathcal{C}_1$, $\hat{x}$ are binary products on $\mathcal{C}_2$ and G preserves those products. Condition 2 instantiates to: Given such α as above, such that F preserves binary products, then H preserves binary products.

3 Rezk Completions of Topoi

In this section, we apply Lemma 1 to lift the Rezk completion to topoi. Recall that an elementary topos is a finitely complete cartesian closed category equipped with a subobject classifier. A morphism of elementary topoi is a functor which preserves each of the structures involved. We denote the bicategory of elementary topoi as ElTop, and the bicategory of topoi whose underlying category is univalent is denoted $\mathsf{ElTop}_{\mathsf{univ}}$. In the formalization, these bicategories are constructed by stacking different displayed bicategories, starting with Cat, as depicted in the following diagram:



The main result establishes that topoi admit Rezk completions:

Theorem 3. *The inclusion* $\mathsf{ElTop}_{\mathsf{univ}} \to \mathsf{ElTop}$ *admits a left biadjoint.*

To prove Theorem 3, we apply Lemma 1 for each of the intermediary displayed bicategories. In particular, we construct Rezk completions for categories equipped with the structures mentioned above. The main challenge in constructing the left biadjoints is transporting the structure on a category along a weak equivalence, which is the first condition in Lemma 1. In particular, the univalence of categories ensures that we can apply the essential surjectiveness of the weak equivalence. The proof strategy for each of these structures follows the same steps as in Example 2.

We expect our method to apply to other kinds of structure. First, a topos possesses numerous structures, such as regularity and exactness, and local cartesian closedness, etcera. Second, there are structures only shared by some topoi, such as the existence of a natural numbers object. Third, there are other "structured categories" which are not topoi, but for which the same strategy can also be applied, e.g., abelian categories.

However, Lemma 1 cannot be applied to construct Rezk completions for structures like monoidal categories [7] and enriched categories [5]. Indeed, the contractibility assumption on the 2-cells is not generally satisfied in these cases since the natural transformations need to commute with the additional structure. Nonetheless, one could try to prove a more general lemma that does apply.

A theoretical obstacle however, is that our framework operates under a crucial assumption: that the "correct notion of univalence" for the structure in question coincides with the univalence of the underlying category. This assumption breaks down for both dagger categories, and enriched categories whose base of enrichment are not assumed to be univalent.

References

[1] B. Ahrens, D. Frumin, M. Maggesi, N. Veltri, and N. van der Weide. Bicategories in univalent foundations. *Math. Struct. Comput. Sci.*, 31(10):1232–1269, 2021.

[2] B. Ahrens, K. Kapulkin, and M. Shulman. Univalent categories and the Rezk completion. *Math. Struct. Comput. Sci.*, 25(5):1010–1039, 2015.

[3] J. M. E. Hyland, P. T. Johnstone, and A. M. Pitts. Tripos theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 88(2):205–232, 1980.

[4] A. M. Pitts. Tripos theory in retrospect. *Math. Struct. Comput. Sci.*, 12(3):265–279, 2002.

[5] N. van der Weide. Univalent enriched categories and the enriched Rezk completion. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, volume 299 of *LIPIcs*, pages 4:1–4:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[6] V. Voevodsky, B. Ahrens, D. Grayson, et al. UniMath — a computer-checked library of univalent mathematics. Available at http://unimath.github.io/UniMath/, 2022.

[7] K. Wullaert, R. Matthes, and B. Ahrens. Univalent monoidal categories. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France*, volume 269 of *LIPIcs*, pages 15:1–15:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

Type Theory and Themes in Philosophical Logic

Greg Restall

Arché Research Centre, Philosophy Department,
The University of St Andrews · gr69@st-andrews.ac.uk

Type theorists share interests and concerns with philosophical logicians. This isn't surprising, since Martin-Löf is (among other things) a philosopher, and type theory was born in philosophy [46, 47, 48, 49]. Although type theory has come into its own in computer science—and, more recently, in mathematics with the rise of assistants—the connections between type theory and philosophical logic go beyond Martin-Löf's original motivations. There are many fresh points of contact with active research areas in philosophical logic. In the interest of fostering communication between these different traditions, I will sketch some of these connections.

Modal and Substructural Logics. Modal logics (extending propositional logic with modal operators, like $\Box$, for necessity, and $\Diamond$ for possibility) became a focus in philosophical logic in the second half of the 20th Century [32, 36], while its application to computer science took some time, with the development of dynamic logic [29, 63, 64]. Substructural logics, on the other hand, arose independently inside philosophy [2, 3, 72] with the study of relevant logics and entailment, linguistics [38, 39, 40, 54, 55] with the Lambek calculus, and computer science [27, 28, 80, 81], in linear logic. Recent work on modal [10, 30, 34] and substructural [45, 53] type theories provide natural areas of intersection with contemporary philosophical logic. The vast bulk of formal work in modal logic uses possible worlds semantics [11, 14, 15], as does philosophical work on substructural logics [43, 44, 59, 67]. These models have their use to represent *propositions* (types) and the entailment between them, but provide little, to no insight concerning the identity of the *proofs* or *constructions* that bear those types, so they can be of only limited use in modelling a properly rich type theory. The same goes for work in *algebraic* models.[1]

Modal and substructural *algebras*, on the other hand, are well understood structures [22, 25, 58] which should prove useful for type-theoretic considerations. One important insight has been the centrality of *residuation* (Galois connection, adjunction) as a unifying principle [18, 21, 22, 66]. A necessity modality $\Box$ gains its distinctive features in connection connected with dual, *possibility* modality $\blacklozenge$, for which we have $a \leq \Box b$ iff $\blacklozenge a \leq b$. Work on the proof theory of modal and substructural logics [7, 62] can provide a more natural point of connection with categorical semantics for type theories [41].

Intensionality and Identity. Homotopy type theory [70, 79] and cubical type theory [4, 5, 16] bring the logic of *identity* into focus, by raising the prospect of different grounds for an identity fact of the form $a = b$. Questions about how best to model identity in a type theory raises questions that philosophers ask using the terms *sense* and *reference*. If $a = b$ then the two different terms a and b must have the same *reference* (or value), but the possibility remains open that these two terms might have different *senses* (meanings). The *extensional* theory of identity is straightforward: everything is identical to itself, and not to anything else [42, p. 192]. Once we move from reference to *sense*, and consider non-extensional phenomena, such as *meaning*, *knowledge*, *proof*, or *construction*, matters are more nuanced [20, 24, 50]. We might

[1] These are partially ordered structures $a \leq b$ iff a entails b. This is a degenerate category in which there is at most one arrow between any two objects.

know that Clark Kent is Clark Kent, without knowing that Clark Kent is Superman, even though Clark Kent *is* Superman. The terms 'Clark Kent' and 'Superman' refer to the same item, but do so in different ways [56, 57]. Different accounts of the semantics of identity give us different options for understanding *how* terms might pick out their values, and, more generally the space of possible semantic values of identity claims between items of each type [9].

Classical Logic. Intuitionistic type theory is constructive, and this raises the question of the status of classical reasoning. There are different approaches to relating classical and constructive logic, such as embedding classical reasoning *inside* a constructive language by way of a translation [78, Sect. 2.3] (which can be interpreted by way of continuations [69, 77]), or extending the term vocabulary [60, 61] or by extending judgement forms to include positive and negative judgements [17, 19]. These approaches parallel considerations in philosophical logic. Some approaches to classical logic start with intuitionist natural deduction and add new inference forms [51, 52], others are *bilateral* [65], encoding proofs involving both positive and negative judgement forms [73, 74, 75]. Other approaches interpret the sequent calculus in terms of assertion and denial [71, 68]. Recent work on assertion and denial distinguishes two forms: *strongly* deny p is to rule p out; to *weakly* deny it is to withdraw its assertion and to keep open the option to strongly deny it [33]. Weak and strong denial both clash with assertion[2] and when treating assertion and denial, it is important to distinguish their strong and weak forms.

Speech Acts. A type theory is an account of *judgement*. One distinctive features of *dependent* type theory is that the rules governing different concepts can interleave the different judgement forms. Whether $B(a)$ counts as a type may depend on whether the term a inhabits another type A. To focus on propositions, whether B counts as a prop can depend on whether another proposition A is true.[3] Traditional grammars form the syntax of the language first, independently of truth conditions, but some contemporary theories of propositional content mirror this structure. 'The king of France is bald' expresses an assertion only if there is a King of France [76]: predication *presupposes* reference. Dependent type theories form a natural context in which presupposition phenomena like this can be modelled and studied.

However, we can do more with our language than form assertions, denials, suppositions and inferences [35]. In natural and in artificial languages, we find imperatives, promises, requests, etc., which differ from judgement forms in many ways [6, 8]. Philosophers and linguists have done a great deal of work on the function and logic different forms of speech acts [23, 26, 37], which may prove salient when exploring the semantics of languages with imperatives, and other non-assertoric forms.

Formal and Applied Theory. Beneath these points of contact, there is a deeper connection between *applied* computational type theory and philosophical logic. A formal type theory is a structural presentation of forms of judgement, which may be interpreted, as Martin-Löf showed us, as a theory of sets, of computation, of propositions, and in other ways besides. A properly *computational* type theory takes the inductive presentation of types and terms to stand atop a fundamental computational substrate [1, 31, 47]. Terms describe *computations*, which are classified by types. The distinction between a formal and an applied type theory parallels

[2] And dually, we have strong assertion (ruling p *in*) and weak assertion (which merely withdraws the strong denial of p and leaves the possibility of strongly asserting p open).

[3] A formation rule for the conditional $\supset$ states that $A \supset B$ is a proposition when A is a proposition and requires that B is a proposition *only given that* A *is true* [49].

Brandom's distinction between a formal and a material account of inference [12, 13]. A properly *material* theory of inference describes the inferential connections between the judgements implicit in our everyday practice. As I learn the concept 'square' I learn that squares are not circles, and that squares have four sides, of equal length. These are not learned as *facts* we articulate, but as capacities we exercise. To learn a language is to learn how different concepts bear upon each other and on the world. Rules for logical concepts enable us to *make explicit* these inferential relations between our more basic commitments, in that the rules governing those concepts tie them to the underlying and preexisting communicative practice. We take justification for a conditional claim $A \to B$ to be provided by the means to justify B in a context where we take A as given. The formal structure of a material theory inference bears a remarkable resemblance to the computational theory of types, where instead of the norms governing justifications and grounds for human judgements, we have computations and their classification into different types. So, it is not surprising that insights from one area can be applied to the other, since there is a single formal framework that describes both domains.

A fresh challenge for research, however, is to develop a properly *hybrid* type theory, encompassing both computation and human communicative practices, in order to better understand the possibilities for communication involving both human judgement and machine computation. After all, two domains of application for the discipline of type theory are in (a) the design of expressive and performant dependently typed functional programming languages, and (b) the design of modular, expressive and natural proof assistants. Both of these tasks involve taking both the *computational* and the *communicative* roles of the underlying type theory seriously, so it seems appropriate to adopt a framework that helps us keep both roles in view at once.[4]

References

[1] S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[2] Alan R. Anderson and Nuel D. Belnap. *Entailment: The Logic of Relevance and Necessity*, volume 1. Princeton University Press, Princeton, 1975.

[3] Alan Ross Anderson, Nuel D. Belnap, and J. Michael Dunn. *Entailment: The Logic of Relevance and Necessity*, volume 2. Princeton University Press, Princeton, 1992.

[4] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. Syntax and models of Cartesian cubical type theory. *Mathematical Structures in Computer Science*, 31(4):424–468, 2021.

[5] Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher-dimensional type theory. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 680–693, New York, NY, USA, 2017. Association for Computing Machinery.

[6] J. L. Austin. *How to do things with words*. Clarendon Press, Cambrige, 1962.

[7] Nuel Belnap. Display logic. *Journal of Philosophical Logic*, 11:375–417, 1982.

[8] Nuel Belnap. Declaratives are not enough. *Philosophical Studies*, 59(1):1–30, 1990.

[9] Nuel Belnap. Under Carnap's lamp: Flat pre-semantics. *Studia Logica*, 80(1):1–28, 2005.

[10] Lars Birkedal. Developing theories of types and computability via realizability. *Electronic Notes in Theoretical Computer Science*, 34:2, 2000.

[11] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001.

[12] Robert B. Brandom. *Making It Explicit*. Harvard University Press, 1994.

[4] Thanks to the referees for helpful comments on the first draft of this abstract.

[13] Robert B. Brandom. *Articulating Reasons: an introduction to inferentialism*. Harvard University Press, 2000.

[14] Alexander Chagrov and Michael Zakharyaschev. *Modal Logic*, volume 35 of *Oxford Logic Guides*. Oxford University Press, 1997.

[15] Brian F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, Cambridge, 1980.

[16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[17] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP '00: Proceedings of the fifth* ACM SIGPLAN *international conference on Functional programming*, pages 233–243, New York, NY, USA, 2000. ACM.

[18] David Darais and David Van Horn. Constructive Galois connections. *Journal of Functional Programming*, 29, 2019.

[19] Paul Downen and Zena M. Ariola. Compiling with classical connectives. *Logical Methods in Computer Science*, 16(3), 07 2019.

[20] Michael Dummett. Sense and reference from a constructivist standpoint. *The Bulletin of Symbolic Logic*, 27(4):485–500, 2021.

[21] J. Michael Dunn. Gaggle theory: An abstraction of Galois connections and residuation with applications to negation and various logical operations. In *Logics in AI, Proceedings European Workshop* JELIA *1990*, volume 478 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[22] J. Michael Dunn and Gary M. Hardegree. *Algebraic Methods in Philosophical Logic*. Clarendon Press, Oxford, 2001.

[23] Daniel Fogal, Daniel W. Harris, and Matt Moss, editors. *New work on speech acts*. Oxford University Press, Oxford, United Kingdom, 2018.

[24] Gottlob Frege. Sense and reference. *The Philosophical Review*, 57(3):209–230, 1948.

[25] Nikolaos Galatos. *Residuated Lattices: An Algebraic Glimpse at Substructural Logics*. Elsevier, Amsterdam; Boston, 2007.

[26] Bart Geurts. Communication as commitment sharing: speech acts, implicatures, common ground. *Theoretical Linguistics*, 45(1-2):1–30, 2019.

[27] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–101, 1987.

[28] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[29] Robert I. Goldblatt. *Logics of Time and Computation*. Number 7. CSLI Publications, 1987.

[30] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[31] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, 2016.

[32] G. Hughes and M. Cresswell. *An Introduction to Modal Logic*. Methuen, London, 1968.

[33] Luca Incurvati and Julian J. Schlöder. *Reasoning With Attitude: Foundations and Applications of Inferential Expressivism*. Oxford University Press, New York, 2023.

[34] G. A. Kavvos and Daniel Gratzer. Under lock and key: A proof system for a multimodal logic. *The Bulletin of Symbolic Logic*, 29(2):264–293, 2023.

[35] John T. Kearns. Conditional assertion, denial, and supposition as illocutionary acts. *Linguistics and Philosophy*, 29(4):455–485, 2006.

[36] Saul A. Kripke. A completeness theorem in modal logic. *The Journal of Symbolic Logic*, 24(1):1–14, 1959.

[37] Rebecca Kukla and Mark Lance. *Yo! and Lo!: The Pragmatic Topography of the Space of Reasons*. Harvard University Press, 2009.

[38] Joachim Lambek. Deductive systems and categories I: Syntactic calculus and residuated categories. *Mathematical Systems Theory*, 2:287–318, 1968.

[39] Joachim Lambek. Deductive systems and categories II: Standard constructions and closed categories. In Peter Hilton, editor, *Category Theory, Homology Theory and their Applications*, number 86 in Lecture Notes in Mathematics, pages 76–122. Springer-Verlag, 1969.

[40] Joachim Lambek. Deductive systems and categories III: Cartesian closed categories, intuitionist propositional calculus, and combinatory logic. In Bill Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, number 274 in Lecture Notes in Mathematics, pages 57–82. Springer-Verlag, 1972.

[41] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23(3-4):281–296, 1969.

[42] David K. Lewis. *On the Plurality of Worlds*. Blackwell, Oxford, 1986.

[43] Edwin Mares. Relevant logic and the theory of information. *Synthese*, 109(3):345–360, 1997.

[44] Edwin D. Mares. *Relevant Logic: A Philosophical Interpretation*. Cambridge University Press, 2004.

[45] Danielle Marshall and Dominic Orchard. Non-linear communication via graded modal session types. *Information and Computation*, 301:105234, 2024.

[46] Per Martin-Löf. *Notes on Constructive Mathematics*. Almqvist and Wiksell, Stockholm, 1970.

[47] Per Martin-Löf. Constructive mathematics and computer programming. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 312(1522):501–518, 1984.

[48] Per Martin-Löf. *Intuitionistic Type Theory: Notes by Giovanni Sambin of a Series of Lectures Given in Padua, June 1980*. Number 1 in Studies in Proof Theory. Bibliopolis, Naples, 1984.

[49] Per Martin-Löf. On the meaning of the logical constants and the justification of the logial laws. *Nordic Journal of Philosophical Logic*, 1:11–69, 1996.

[50] Per Martin-Löf. The sense/reference distinction in constructive semantics. *The Bulletin of Symbolic Logic*, 27(4):501–513, 2021.

[51] Peter Milne. Classical harmony: rules of inference and the meaning of the logical constants. *Synthese*, 100:49–94, 1994.

[52] Peter Milne. Harmony, purity, simplicity and a 'seemingly magical fact'. *Monist*, 85(4):498–534, 2002.

[53] Benjamin Moon, Harley Eades, and Dominic Orchard. Graded modal dependent type theory. In *Programming Languages and Systems: 30th European Symposium on Programming, Esop 2021*. Springer International Publishing, 2021.

[54] Michael Moortgat. *Categorial Investigations: Logical Aspects of the Lambek Calculus*. Foris, Dordrecht, 1988.

[55] Glyn Morrill. *Type Logical Grammar: Categorial Logic of Signs*. Kluwer, Dordrecht, 1994.

[56] Yiannis N. Moschovakis. *Sense and denotation as algorithm and value*, volume Volume 2 of *Lecture Notes in Logic*, pages 210–249. Springer-Verlag, Berlin, 1993.

[57] Reinhard Muskens. Sense and the computation of reference. *Linguistics and Philosophy*, 28(4):473–504, 2005.

[58] Hiroakira Ono. *Proof Theory and Algebra in Logic*. Springer, Aug 2019.

[59] Francesco Paoli. *Substructural Logics: A Primer*. Springer, May 2002.

[60] Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 190–201. Springer, 1992.

[61] Michel Parigot. Classical proofs as programs. In George Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Computational Logic and Proof Theory*, volume 713 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 1993.

[62] Francesca Poggiolesi. *Gentzen Calculi for Modal Propositional Logic*. Trends in Logic. Springer, 2010.

[63] V. R. Pratt. Semantical considerations on Floyd–Hoare logic. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[64] V. R. Pratt. Dynamic algebras and the nature of induction. In *12th ACM Symposium on the Theory of Computation*, Los Angeles, 1980.

[65] Huw Price. Why 'not'? *Mind*, 99(394):222–238, 1990.

[66] Greg Restall. Display logic and gaggle theory. *Reports in Mathematical Logic*, 29:133–146, 1995.

[67] Greg Restall. *An Introduction to Substructural Logics*. Routledge, 2000.

[68] Greg Restall. Multiple conclusions. In Petr Hájek, Luis Valdés-Villanueva, and Dag Westerståhl, editors, *Logic, Methodology and Philosophy of Science: Proceedings of the Twelfth International Congress*, pages 189–205. KCL Publications, 2005.

[69] John C. Reynolds. The discoveries of continuations. LISP *and Symbolic Computation*, 6(3-4):233–247, 1993.

[70] Egbert Rijke. Introduction to homotopy type theory. ArXiV 2212.11082, 2022. Prepublication version. To be published by *Cambridge University Press*.

[71] David Ripley. Bilateralism, coherence, warrant. In Friederike Moltmann and Mark Textor, editors, *Act-Based Conceptions of Propositional Content*, pages 307–324. Oxford University Press, Oxford, UK, 2017.

[72] Richard Routley and Robert K. Meyer. Semantics of entailment. In Hugues Leblanc, editor, *Truth, Syntax and Modality*, pages 194–243. North Holland, 1973. Proceedings of the Temple University Conference on Alternative Semantics.

[73] Ian Rumfitt. "Yes" and "No". *Mind*, 109(436):781–823, 2000.

[74] Ian Rumfitt. *The Boundary Stones of Thought: An Essay in the Philosophy of Logic*. Oxford University Press, 2015.

[75] Timothy Smiley. Rejection. *Analysis*, 56:1–9, 1996.

[76] P. F. Strawson. On referring. *Mind*, 59(235):320–344, 1950.

[77] Th. Streicher and B. Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

[78] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, second edition, 2000.

[79] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[80] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[81] David Walker. Substructural type systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–44. MIT Press, 2004.

Constructive algebraic completeness of first-order bi-intuitionistic logic

Dominik Kirst[1] and Ian Shillito[2]

[1] Université Paris Cité, IRIF, Inria, Paris, France
dominik.kirst@inria.fr
[2] University of Birmingham, Birmingham, United Kingdom
i.b.p.shillito@bham.ac.uk

Background Bi-intuitionistic logic extends intuitionistic logic with the *exclusion* binary operator $-\!\!<$, dual to implication. This extension is mathematically natural, as symmetry is regained in the language: each operator has a dual, even $\neg\varphi := \varphi \to \bot$ has $\sim\varphi := \top -\!\!< \varphi$. However, bi-intuitionistic logic is packed with surprises. First, it proves a bi-intuitionistic version of LEM ($\varphi \vee \sim\varphi$), and is thus not constructive as it fails the disjunction property. Second, despite not being constructive it does not yet collapse to classical logic, given that it is a conservative extension of intuitionistic logic (in the propositional case). Third, this logic bears a striking resemblance to modal logic, as it splits into a *local* and *global* logic[1] and is tightly connected to the tense logic S4t [19, 20, 21]. Finally, the conservativity of bi-intuitionistic logic over intuitionistic logic only holds in the propositional case: the *constant domain axiom* $(\forall x(\varphi(x) \vee \psi) \to (\forall x\varphi(x) \vee \psi))$ is provable in *first-order* bi-intuitionistic logic, but not intuitionistically.

Historically, bi-intuitionistic logic was developed by Cecylia Rauszer in a series of articles in the 1970s [25, 24, 26, 27] leading to her Ph.D. thesis [28].[2] She studied this logic under a wide variety of aspects: algebraic semantics, axiomatic calculus, sequent calculus, and Kripke semantics. Unfortunately, through time a variety of mistakes have been detected in her work. First, Crolard proved in 2001 that bi-intuitionistic logic is not complete for the class of rooted frames [2, Corollary 2.18], in contradiction with Rauszer's proof which relies on rooted canonical models. Second, Pinto and Uustalu found in 2009 [18] a counterexample to Rauszer's claim of admissibility of cut for the sequent calculus she designed [24, Theorem 2.4]. Finally, Goré and Shillito in 2020 [7] detected confusions in Rauszer's work about the holding of the deduction theorem in bi-intuitionistic logic, as well as issues in her completeness proofs.

The foundations of bi-intuitionistic logic have therefore been in reconstruction since these discoveries. In the propositional case serious progress has been made: sequent calculi for the local logic were provided [18]; axiomatic calculi and Kripke semantics were connected [7, 29] and reverse analysed in a constructive setting [30]; the algebraic treatment was recently completed by Deakin and Shillito [4, 3]. As for the first-order case, it has received little attention until recently, where it was shown to fail Craig interpolation [17], studied via polytree labelled sequent calculi [13], and its soundness and completeness w.r.t. its Kripke semantics were established [10].

No reconstruction of the algebraic treatment of first-order bi-intuitionistic logic has yet been provided. We fill this gap by proving that the local first-order bi-intuitionistic logic is *sound* and *weakly complete* with respect to bi-Heyting algebras: $\vdash \varphi$ iff $\vDash \varphi$. Our work is motivated by foundational interests, but also by constructive sensitivity: these algebraic results usually hold constructively (cf. [5]), while our previous proofs [10] for the Kripke semantics use classical principles and certainly necessarily so, given the already established propositional analysis [30].

[1] This terminology comes from the Kripke semantics, in which one can define a local and a global notion of semantic consequence [1, Section 1.5].

[2] While appearances of bi-intuitionistic can be detected prior to these articles, notably in 1942 in Moisil's work [15], in 1964 in Grzegorczyk's work [8], and in 1971 in Klemke's work [11], the breadth and foundational nature of Rauszer's work advocate for her place as founder of this logic.

Algebraic interpretation of quantifiers The algebraic interpretation of the propositional bi-intuitionistic connectives straightforwardly extends the one for intuitionistic logic: Heyting algebras, i.e. bounded distributive lattices with an implication satisfying the residuation on the left below, are extended to bi-Heyting algebras with the exclusion operator satisfying the dual residuation on the right.

$$\frac{\varphi \wedge \psi \leq \chi}{\varphi \leq \psi \to \chi} \qquad\qquad \frac{\varphi \leq \psi \vee \chi}{\varphi \multimap \psi \leq \chi}$$

However, the interpretation of quantifiers in algebraic semantics is famously not obvious. Some approaches involve complex algebraically inspired structures, like hyperdoctrines [12] or cylindric algebras [16]. Instead, we follow a tradition [23, 22] of grafting interpretation of quantifiers on the base algebras (bi-Heyting in our case) via the creation of *infima* $\bigsqcap$, interpreting $\forall$, and *suprema* $\bigsqcup$, interpreting $\exists$. The strikingly obvious issue with this approach is that some bi-Heyting algebras do not have such infinite elements. We simply discard these algebras and focus on the class of *complete* bi-Heyting algebras, i.e. those possessing these infinite elements.

MacNeille completion of the Lindenbaum-Tarski algebra Traditionally, one proves (weak) completeness by building a so-called Lindenbaum-Tarski (LT) algebra for the logic, a syntactic algebra made out of equivalence classes of formulas $[\![\varphi]\!] := \{\psi \mid \vdash \varphi \leftrightarrow \psi\}$ in which $\leq$ captures entailment: $[\![\varphi]\!] \leq [\![\psi]\!]$ iff $\varphi \vdash \psi$. This algebra is easily defined for propositional bi-intuitionistic logic, constituting the ideal candidate to graft quantifiers onto to prove completeness. Sadly, we are not ensured that this algebra is complete, hence it may then be outside of the class of complete algebras under focus. To overcome this difficulty, we complete the LT algebra via the *MacNeille completion* [14], which embeds a partially ordered set $(X, \leq)$ into the complete lattice of all its subsets A which are *successively* closed under upper bounds and lower bounds ($(A^u)^l \subseteq A$). In the resulting algebra, $\to$ receives a natural interpretation giving rise to a Heyting algebra [22, 9], leaving us with the task of interpreting $\multimap$ to obtain completeness. As indicated by Harding and Bezhanishvili [9], this interpretation is not natural:

$$X \multimap Y := \{x \mid \forall y \in Y^u.\ (y \vee x) \in X^u\}^l$$

Still, we obtain a complete bi-Heyting algebra embedding the initial propositional LT algebra, establishing weak completeness.

A Pathway to strong completeness Ideally, we would like to prove *strong* completeness, i.e. $\Gamma \vdash \varphi$ iff $\Gamma \vDash \varphi$, with respect to the following algebraic consequence relation, where φ^* is the interpretation of φ in A via the valuation V mapping atomic formulas to elements of X.

$$\Gamma \vDash \varphi \qquad := \qquad \forall A.\ \forall V.\ \forall a \in A.\quad (\forall \gamma \in \Gamma.\ a \leq \gamma^*) \quad \to \quad a \leq \varphi^*$$

To prove strong completeness, we want to generate a proof of $\Gamma \vdash \varphi$ from $\Gamma \vDash \varphi$ using the completed LT algebra. It would then suffice to find an element below the interpretation of all $\gamma \in \Gamma$. An obvious candidate is the (existing) infimum $\bigsqcap \Gamma^*$, which is below φ^* under the assumption $\Gamma \vDash \varphi$. Unfortunately, $\bigsqcap \Gamma$ is not a formula in our language, so we cannot extract $\bigsqcap \Gamma \vdash \varphi$ from $\bigsqcap \Gamma^* \leq \varphi^*$. If the infimum of Γ^* was *compact*, i.e. such that $\bigsqcap \Gamma^* \leq a$ entails the existence of a *finite* subset $\Delta^* \subseteq \Gamma^*$ with $\bigsqcap \Delta^* \leq a$, we would be able to close the strong completeness proof. A way to obtain compact infima and suprema is through *canonical extensions* [6]. In further work we will inspect the interplay between MacNeille completion for bi-Heyting algebras and canonical extension, in the hope to deduce strong completeness.

Acknowledgments We thank Marco Abbadini for his help in the finding of the interpretation of exclusion in the MacNeille-completed LT algebra.

References

[1] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[2] Tristan Crolard. Subtractive logic. *TCS*, 254:1-2:151–185, 2001.

[3] Jonte Deakin and Shillito Ian. Bi-intuitionistic logics through the abstract algebraic logic lens. Unpublished, to appear.

[4] Jonte Deakin and Shillito Ian. Weak and strong bi-intuitionistic logics from an abstract algebraic logic perspective. In *Australasian Association for Logic 2024 Conference*, 2024.

[5] Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory: Extended version. *Journal of Logic and Computation*, 31(1):112–151, 2021.

[6] Mai Gehrke. *Canonical Extensions, Esakia Spaces, and Universal Models*, pages 9–41. Springer Netherlands, Dordrecht, 2014.

[7] Rajeev Goré and Ian Shillito. Bi-Intuitionistic Logics: A New Instance of an Old Problem. In *Advances in Modal Logic 13, papers from the thirteenth conference on "Advances in Modal Logic," held online, 24-28 August 2020*, pages 269–288, 2020.

[8] Andrzej Grzegorczyk. A philosophically plausible formal interpretation of intuitionistic logic. *Indagationes Mathematicae*, 26:596–601, 1964.

[9] John Harding and Guram Bezhanishvili. Macneille completions of heyting algebras. *Houston Journal of Mathematics*, 30:937–952, 01 2004.

[10] Dominik Kirst and Ian Shillito. Completeness of first-order bi-intuitionistic logic. In Jörg Endrullis and Sylvain Schmitz, editors, *33rd EACSL Annual Conference on Computer Science Logic, CSL 2025, February 10-14, 2025, Amsterdam, Netherlands*, volume 326 of *LIPIcs*, pages 40:1–40:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[11] Dieter Klemke. Ein Henkin-Beweis für die Vollständigkeit eines Kalküls relativ zur Grzegorczyk-Semantik. *Archiv für mathematische Logik und Grundlagenforschung*, 14:148–161, 1971.

[12] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23(3/4):281–296, 1969.

[13] Tim S. Lyon, Ian Shillito, and Alwen Tiu. Taking bi-intuitionistic logic first-order: A proof-theoretic investigation via polytree sequents. In Jörg Endrullis and Sylvain Schmitz, editors, *33rd EACSL Annual Conference on Computer Science Logic, CSL 2025, February 10-14, 2025, Amsterdam, Netherlands*, volume 326 of *LIPIcs*, pages 41:1–41:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[14] H.M. MacNeille. Partially ordered sets. *Transactions of the American Mathematical Society*, 42:416–460, 1937.

[15] C. Moisil. Logique modale. *Disquisitiones malhematicae et physicae*, 2:3–98, 1942.

[16] J. Donald Monk. *Cylindric Algebras*, pages 219–229. Springer New York, New York, NY, 1976.

[17] Grigory K Olkhovikov and Guillermo Badia. Craig interpolation theorem fails in bi-intuitionistic predicate logic. *The Review of Symbolic Logic*, 17(2):611–633, 2024.

[18] Luís Pinto and Tarmo Uustalu. Proof search and counter-model construction for bi-intuitionistic propositional logic with labelled sequents. In M. Giese and A. Waaler, editors, *Proc. TABLEAUX*, pages 295–309, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[19] Arthur N. Prior. *Time and Modality*. Greenwood Press, Westport, Conn., 1955.

[20] Arthur N. Prior. *Past, Present and Future*. Clarendon P., Oxford,, 1967.

[21] Arthur N. Prior. *Papers on Time and Tense*. Oxford University Press UK, Oxford, England, 1968.

[22] H. Rasiowa. Algebraic treatment of the functional calculi of heyting and lewis. *Fundamenta Mathematicae*, 38(1):99–126, 1951.

[23] H. Rasiowa and Roman Sikorski. A proof of the completeness theorem of gödel. *Fundamenta Mathematicae*, 37(1):193–200, 1950.

[24] C. Rauszer. A formalization of the propositional calculus of h-b logic. *Studia Logica*, 33:23–34, 1974.

[25] C. Rauszer. Semi-boolean algebras and their applications to intuitionistic logic with dual operations. *Fundamenta Mathematicae*, 83:219–249, 1974.

[26] C. Rauszer. On the strong semantical completeness of any extension of the intuitionistic predicate calculus. *Studia Logica*, 33:81–87, 1976.

[27] C. Rauszer. The craig interpolation theorem for an extension of intuitionistic logic. *Journal de l'Académie Polonaise des Sciences*, 25:127–135, 1977.

[28] Cecylia Rauszer. *An Algebraic and Kripke-Style Approach to a Certain Extension of Intuitionistic Logic*. PhD thesis, Instytut Matematyczny Polskiej Akademi Nauk, 1980.

[29] Ian Shillito. *New Foundations for the Proof Theory of Bi-Intuitionistic and Provability Logics Mechanized in Coq*. PhD thesis, Australian National University, Canberra, 2023.

[30] Ian Shillito and Dominik Kirst. A mechanised and constructive reverse analysis of soundness and completeness of bi-intuitionistic logic. In Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, pages 218–229. ACM, 2024.

Higher-Order Focusing on Linearity and Effects

Siva Somayyajula

ssomayya@alumni.cmu.edu

Abstract

The relationship between effect calculi and focused logics has been extensively studied through their shared notion of polarization. We contribute another data point by extending Zeilberger's higher-order focused logic of delimited continuations to subsume the type structure of the enriched effect calculus. Then, we report ongoing work on modelling the linear usage of state via the interaction between the linear state monad and linear lenses.

Polarized calculi arise both as internal languages to adjunction models of effects [5] and as term assignments to focused logics [3]. The former, involving an adjunction between two categories, assigns a polarity to a type by the category from which it originates. In the latter, polarization arises from the assortation of logical rules on the basis of (non)invertibility, i.e., whether their application during proof search may require backtracking. The correspondence in type structure typically extends to terms and equational theories—for example, see Rioux and Zdancewic [20].

To phrase the question of interest, stated in the next paragraph, let us spell out the relationship between call-by-push-value (CBPV) [11] and focused intuitionistic logic [13, 21]. From the point of view of focusing, positive and negative types are allowed to respectively be right- and left-noninvertible. Dually, they *must* respectively be left- and right-invertible. This gives rise to four judgments: either inversion or focus on the left- or right-hand side of the sequent, each concerned with the iterated application of invertible or noninvertible rules. Modulo antecedent polarity [10], non-complex values and computations correspond to right focus resp. inversion terms. Non-complex stacks arising from the CK-machine semantics of CBPV [12] coincide with the left focus terms from *weakly* focused [22] intuitionistic logic.

Linearity in the enriched effect calculus (EEC) [7] arises by internalizing complex stacks via the linear function space, i.e., by reading a stack as a computation homomorphism. Unfortunately, the previous analogy appears to break down with the EEC when naively presented as a sequent calculus: the value types of (non)linear functions are not left-invertible. Moreover, the copower and computational sum types are not right-invertible. One solution is to decouple linearity from effects as Curien et al. [5] do with their effect calculi, which remain faithful to the focusing properties of intuitionistic linear logic, and whose models subsume those of the EEC. We approach the problem from a different angle: is there a judgmental reformulation of focused intuitionistic logic that can be *retrofitted* with linearity as conceived by the EEC?

Zeilberger lays the groundwork for a positive answer with the *higher-order* focused logic [23, 25] of delimited continuations [26]. Under higher-order focusing, inversion terms are metalevel maps out of focus terms. Not only are the derived logical rules trivially invertible, but admissibility of cut and identity is immediate. From there, our contribution is to "backpatch" into the logic the EEC connectives in question. Formally, fix right and left focus judgments $[P]$ and $N > P$ where P and N are positive and negative types. Then, the inversion judgments $\Gamma \vdash P$ and $\Gamma \vdash N$ are defined by the top row of inference rules in Figure 1 using the metalevel function space ($\Rightarrow$). In particular, a right inversion term is a stack-passing *value* parametric in the positive answer type; note that a negative answer type as in weak focusing would produce a circular definition. The formerly "problematic" connectives are given logical rules in the second and third rows of the same figure, renaming the positive ($\to$) to ($\supset$), I to $\mathbf{1}$ (not to be confused with 1, the positive unit), ($+$) to ($\oplus$), and ($\oplus$) to ($\parr$).

$$
\begin{aligned}
\text{index sets } & S = \{\ell, k, \ldots\} \\
\text{positive types } & P, Q := P \supset Q \mid 1 \mid P \times Q \mid \oplus\{\ell : P_\ell\}_{\ell \in S} \mid N \multimap\!\!\bullet\, P \mid N \multimap O \\
\text{negative types } & M, N, O := P \to N \mid \mathbf{1} \mid P \otimes N \mid \parr\{\ell : N_\ell\}_{\ell \in S} \mid \&\{\ell : N_\ell\}_{\ell \in S} \\
\text{types } & A := P \mid N \\
\text{contexts } & \Gamma := \cdot \mid \Gamma, P \\
\text{judgments } & := [P] \mid N > P \mid \Gamma \vdash A \mid N \gg O
\end{aligned}
$$

$$
\frac{[P]}{\cdot \vdash P} \qquad \frac{[P] \Rightarrow \Gamma \vdash Q}{\Gamma, P \vdash Q} \qquad \frac{\text{for all } P\text{: } N > P \Rightarrow \Gamma \vdash P}{\Gamma \vdash N} \qquad \frac{\text{for all } P\text{: } O > P \Rightarrow N > P}{N \gg O}
$$

$$
\frac{[P] \Rightarrow [Q]}{[P \supset Q]}\ \supset\mathrm{R} \qquad \frac{N > P}{[N \multimap\!\!\bullet\, P]}\ \multimap\!\!\bullet\,\mathrm{R} \qquad \frac{N \gg O}{[N \multimap O]}\ \multimap\mathrm{R} \qquad \frac{[P]}{\mathbf{1} > P}\ \mathbf{1}\mathrm{L} \qquad \frac{[P] \Rightarrow N > Q}{P \otimes N > Q}\ \otimes\mathrm{L}
$$

$$
\frac{\{N_\ell > P\}_{\ell \in S}}{\parr\{\ell : N_\ell\}_{\ell \in S} > P}\ \parr\mathrm{L} \qquad \frac{\Gamma \vdash N \quad N > P}{\Gamma \vdash P}\ \mathrm{cut}_{>} \qquad \frac{N \gg O \quad O > P}{N > P}\ \mathrm{cut}^{\gg} \qquad \frac{N \gg M \quad M \gg O}{N \gg O}\ \mathrm{cut}^{-}_{-}
$$

$$
\frac{\Gamma \vdash N \quad N \gg O}{\Gamma \vdash O}\ \mathrm{cut}_{\gg} \qquad \frac{\Gamma \vdash P \quad \Gamma, P \vdash A}{\Gamma \vdash A}\ \mathrm{cut}^{+} \qquad \frac{}{\Gamma, P \vdash P}\ \mathrm{id}^{+} \qquad \frac{}{N \gg N}\ \mathrm{id}^{-}
$$

Figure 1: Types, Judgments, and Selected Rules

The first step toward the linear function space is to note that the positive type ($\multimap\!\!\bullet$), which internalizes stacks, can be read as a generalized negation. To recover the negative codomain, our core observation is to define a new judgment $N \gg O$, internalized by the positive type $N \multimap O$, by applying contraposition, i.e., $N \multimap O \simeq (O \multimap\!\!\bullet\, P) \supset (N \multimap\!\!\bullet\, P)$ parametrically in P. The resulting terms are stack transformers [8] related to the `where` construct of complex stacks but opposite to linear continuation transformers [4].

On the one hand, this calculus subsumes the type structure of the EEC, but also diverges in the term assignment. On the other, we also depart from Zeilberger [26] by omitting explicit pattern variables and have the shift modalities between polarities $\downarrow N \triangleq \mathbf{1} \multimap N$ and $\uparrow P \triangleq P \otimes \mathbf{1}$ be definable rather than be primitive. In terms of metatheory, the expected cut and identity rules are *immediately* admissible—indicated by double lines, they are enumerated in the last two rows. For example, each cut rule amounts to function composition in the metatheory. One goal of this talk is to solicit feedback concerning the denotational and categorical semantics of this calculus; under *defunctionalization*, we review an option in the related work.

One practical and initial motivation of this work concerned tying stacks to the linear usage of state [17] by way of *lenses*. In particular, the type of *linear lenses* [19] $N \rightsquigarrow O$, representing a functional reference of type O within the type N, is definable as $N \multimap ((O \multimap N) \otimes O)$. As do Møgelberg and Staton [17], we can first define the linear state monad in P as $\mathrm{State}_N(P) \triangleq N \multimap P \otimes N$ where N is the state type with the associated operations. From there, we can define a function of type $(N \rightsquigarrow O) \supset \mathrm{State}_O(P) \supset \mathrm{State}_N(P)$ that lifts mutations of an O-state into those of an N-state. However, we unsuccessfully attempted to systematically generate linear lenses by the structure of N, i.e., by converting $N \multimap\!\!\bullet\, P$ to $N \rightsquigarrow \uparrow P$. Looking forward, we are interested in resolving this issue as well as in determining the formal relationship of this calculus to its contemporaries (see below).

Related Work L-calculi [6, 18] and their descendant effect calculi [5] are related to higher-order focusing under defunctionalization [24], dualizing the exposition: reduction is *a priori* and the observation about inversion terms mapping out of focus terms is derived. Thus, it should be possible to determine the exact relationship between the presented calculus and the models of *ops. cit.* Separately, Zeilberger's original presentation [26] included type variables to enforce parametric use of answer types; if internalized, the resulting calculus would be comparable to polymorphic CBPV [20] with first-class stacks and stack manipulation [16, 9]. Lastly, an extension to dependent types would draw a correspondence to eMLTT [2, 1] and other polarized/focused dependent type theories [14, 15].

References

[1] Danel Ahman. Fibred Computational Effects. *CoRR*, abs/1710.02594, 2017.

[2] Danel Ahman, Neil Ghani, and Gordon D. Plotkin. Dependent Types and Fibred Computational Effects. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 36–54, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[3] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 06 1992.

[4] Josh Berdine, Peter O'Hearn, Uday Reddy, and Hayo Thielecke. Linear Continuation-Passing. *Higher-Order and Symbolic Computation*, 15(2/3):181–208, 2002.

[5] Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 44–56, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Pierre-Louis Curien and Guillaume Munch-Maccagnoni. The Duality of Computation under Focus. In Cristian S. Calude and Vladimiro Sassone, editors, *Theoretical Computer Science*, pages 165–181, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[7] Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. The enriched effect calculus: syntax and semantics. *Journal of Logic and Computation*, 24(3):615–654, 06 2012.

[8] Willem Heijltjes. The Functional Machine Calculus. *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 1 - Proceedings of MFPS XXXVIII, Feb 2023.

[9] Yuchen Jiang, Runze Xue, and Max S. New. Notions of Stack-manipulating Computation and Relative Monads (Extended Version), 2025.

[10] Neel Krishnaswami. Focusing is not Call-by-Push-Value, 2014.

[11] Paul Blain Levy. Call-by-Push-Value: A Subsuming Paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 228–243, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[12] Paul Blain Levy. Adjunction Models For Call-By-Push-Value With Stacks. *Electronic Notes in Theoretical Computer Science*, 69:248–271, 2003. CTCS '02, Category Theory and Computer Science.

[13] Chuck Liang and Dale Miller. Focusing and Polarization in Intuitionistic Logic. In *Computer Science Logic*, Lausanne, Switzerland, September 2007.

[14] Daniel R. Licata and Robert Harper. Positively Dependent Types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, page 314, New York, NY, USA, 2009. Association for Computing Machinery.

[15] Étienne Miquey, Xavier Montillet, and Guillaume Munch-Maccagnoni. Dependent Type Theory in Polarised Sequent Calculus. In *TYPES 2020 - 26th International Conference on Types for Proofs and Programs*, pages 1–3, Torino, Italy, March 2020.

[16] Rasmus Ejlers Møgelberg and Alex Simpson. Relational Parametricity for Computational Effects. *Logical Methods in Computer Science*, Volume 5, Issue 3, Aug 2009.

[17] Rasmus Ejlers Møgelberg and Sam Staton. Linear usage of state. *Logical Methods in Computer Science*, Volume 10, Issue 1, Mar 2014.

[18] Guillaume Munch-Maccagnoni and Gabriel Scherer. Polarised Intermediate Representation of Lambda Calculus with Sums. In *Thirtieth Annual ACM/IEEE Symposium on Logic In Computer Science (LICS 2015)*, Kyoto, Japan, July 2015. Dec. 2015: see the added footnote on page 7.

[19] Mitchell Riley. Categories of Optics, 2018.

[20] Nick Rioux and Steve Zdancewic. Computation Focusing. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[21] Robert J. Simmons. Structural Focalization. *ACM Trans. Comput. Logic*, 15(3), September 2014.

[22] Robert J. Simmons and Frank Pfenning. Weak Focusing for Ordered Linear Logic. June 2008.

[23] Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1):66–96, 2008.

[24] Noam Zeilberger. Defunctionalizing Focusing Proofs. In *Proceedings of the 2009 International Workshop on Proof-Search in Type Theories*, PSTT 2009, 2009.

[25] Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, USA, 2009. AAI3358066.

[26] Noam Zeilberger. Polarity and the Logic of Delimited Continuations. In *25th Annual IEEE Symposium on Logic in Computer Science*, pages 219–227, 2010.

Monadic Equational Reasoning for `while` loop in Rocq

Ryuji Kawakami[1], Jacques Garrigue[1], Takahumi Saikawa[1], and Reynald Affeldt[2]

[1] Nagoya University, Japan
[2] National Institute of Advanced Industrial Science and Technology (AIST), Japan

Monadic equational reasoning. Pure functional programs can be reasoned about using equational reasoning thanks to their referential transparency. For programs containing computational effects, Gibbons and Hinze [GH11] proposed monadic equational reasoning, which extends equational reasoning to the verification of programs designed around monads. The interface of each monad is defined as a collection of operators and equations, that allow to manipulate effects.

Monae. Monae [ANS19] is a library that enables verification using monadic equational reasonig in Rocq. Monae consists of interfaces and models. The models guarantee the consistency of the interfaces. Rocq guarantees the correctness of the verification, and the math library MathComp/SSReflect [GM10] makes it possible to write concise proofs. Monae implements a hierarchy of interfaces using Hierarchy Builder [CST20], and allows for the combination of multiple monads and reusable lemmas. Dijkstra monads [SHK+16] also provide an alternative formal framework to verify monadic programs, albeit using Hoare logic rather than equational reasoning.

Non-structurally recursive functions. Proof assistants such as Rocq, which allow for the reduction of programs, do not permit the definition of non-terminating functions to guarantee consistency. Rocq's `Fixpoint` command can only define structurally recursive functions. For non-structural recursion, it is necessary to use an additional accessibility predicate, corresponding to a well-founded order, either directly or through the `Function` or `Equations` commands. For example, McCarthy's 91 function `mc91`, which performs complex recursion, cannot be defined through direct structural recursion in Rocq.

```
let rec mc91 m = if 100 < m then m - 10 else mc91 (mc91 (m + 11))
```

Moreover, functions whose termination is unknown, such as the Collatz predicate, cannot be defined as recursive functions in Rocq.

Defining functions containing `while` statements by a coinductive type. Another way of dealing with non-structurally recursive functions in such proof assistants is to use corecursive definitions, which allow for defining infinite sequences of data. Since Rocq allows an infinite number of constructor applications as long as the guard constraint is satisfied, it is possible to use corecursive definitions to make recursive calls without limit.

The Delay monad proposed by Capretta [Cap05] can be used to represent such corecursive functions as monadic programs. Interaction Trees [SZ21] are a natural generalization of the delay monad to represent non-structurally recursive functions with events. In our work, by defining the interface to the Delay monad as a complete Elgot monad [AMV10], we are able to reason about functions with `while` statements that need not be guaranteed to terminate, such as McCarthy's 91 function and the Collatz predicate.

Complete Elgot monad. The theory for complete Elgot monads corresponds to Iteration theory [BÉ93], which deals with recursive structures algebraically.

In our study, we use the function `while` to define a complete Elgot monad. It is defined using the `CoFixpoint` command, where the right embedded value `inr x` is the continuation of iterations with value `x` and the left embedded value `inl a` is the end of the iteration with the return value `a`.

```
CoFixpoint while {X A} (body: X -> M (A + X)) : X -> M A :=
  fun x => (body x) >>= (fun xa => match xa with
                                   | inr x => DLater (while body x)
                                   | inl a => DNow a end).
```

Uustalu and Veltri [UV17] have shown, by quotienting by an equivalence relation that ignores a finite number of computational steps, that the delay monad is a complete Elgot monad.

Combining computational effects using monad transformers. By using a complete Elgot monad to represent a `while` statement, we can handle functions that contain `while` statements, but only pure functions. For example, a factorial computed using references and `while` statements cannot be expressed using only a complete Elgot monad.

```
let fact n =
  let r = ref 1 in
  let l = ref 1 in



  while !l <= n do

    r := !r * !l;
    l := !l + 1;

  done;
  !r
```

```
Definition factdts n :=
  do r <- cnew ml_int 1;
  do l <- cnew ml_int 1;
  do _ <-
  while (fun (_:unit) =>
         do i <- cget l;
         if i <= n
         then do v <- cget r;
              do _ <- cput r (i * v);
              do _ <- cput l (i.+1);
              Ret (inr tt)
         else Ret (inl tt)) tt;
  do v <- cget r; Ret v.
```

We use monad transformers [AN20] to combine any complete Elgot monad with the exception monad and the typed store monad introduced to represent OCaml references [AGS25, Section 5]. In our work, we show the exception monad transformer, the store monad transformer and the typed store monad transformer preserve the complete Elgot monad structure. This allows verifying programs with multiple effects in Monae.

Contribution. Our contributions are as follows.

1. We define the interface of the delay monad as a complete Elgot monad and show its consistency. This allows Monae to verify functions containing `while` statements.
2. By using monad transformers for combination and the `setoid_rewrite` tactic for generalised rewriting, we have confirmed that verification with Monae is practical for functions involving `while` statements together with other effects.

The code for this work can be found at:

https://github.com/affeldt-aist/monae/pull/147

Acknowledgments. The authors acknowledge support of the JSPS KAKENHI Grants Number 22H00520 and Number 22K11902.

References

[AGS25] Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa. A practical formalization of monadic equational reasoning in dependent-type theory. *Journal of Functional Programming*, 35:e1, 2025.

[AMV10] Jirí Adámek, Stefan Milius, and Jirí Velebil. Equational properties of iterative monads. *Information and Computation*, 208(12):1306–1348, 2010.

[AN20] Reynald Affeldt and David Nowak. Extending equational monadic reasoning with monad transformers. In *26th International Conference on Types for Proofs and Programs (TYPES 2020), March 2–5, 2020, University of Turin, Italy*, volume 188 of *LIPIcs*, pages 2:1–2:21, 2020.

[ANS19] Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019*, volume 11825 of *LNCS*, pages 226–254. Springer, 2019.

[BÉ93] Stephen L. Bloom and Zoltán Ésik. *Iteration Theories—The Equational Logic of Iterative Processes*. EATCS Monographs on Theoretical Computer Science. Springer, 1993.

[Cap05] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[CST20] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), June 29–July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[GH11] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *16th ACM SIGPLAN international conference on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011*, pages 2–14. ACM, 2011.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *Proc. ACM Program. Lang.*, 51(POPL):256–270, 2016.

[SZ21] Lucas Silver and Steve Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. 5(POPL):1–28, 2021.

[UV17] Tarmo Uustalu and Niccolò Veltri. The delay monad and restriction categories. In *14th International Colloquium on Theoretical Aspects of Computing (ICTAC 2017), Hanoi, Vietnam, October 23–27, 2017*, volume 10580 of *Lecture Notes in Computer Science*, pages 32–50. Springer, 2017.

Commuting Rules for the Later Modality and Quantifiers in Step-Indexed Logics

Bálint Kocsis and Robbert Krebbers

Radboud University Nijmegen, The Netherlands
balint.kocsis@ru.nl mail@robbertkrebbers.nl

Abstract

Step-indexing is a semantic tool for stratifying circular, non-wellfounded definitions. The main idea is to use sequences of successive approximations to construct objects such as types, propositions, and functions. It can be formalised in a logical, or type-theoretical, setting, through a modality, called *later*, which allows us to talk about the next approximation.

The internal logic of the topos of trees provides a full-fledged higher-order logic with dependent types and built-in step-indexing. As in any modal logic, it is desirable to have commuting rules for modalities and quantifiers. However, the current known such rules are not satisfactory since they depend on an assumption on the domain of quantification.

We propose novel rules as alternatives from which the former ones can be derived. Our insights are based on the observation that the later modality can be decomposed into two parts. We have formalised our results in the Rocq Prover.

Step-indexing and the topos of trees. Step-indexing is a widely used semantic tool for stratifying circular, non-wellfounded definitions [2], particularly in program logics (*e.g.,* Iris [11]) and logical relation models of type systems with general recursive types [2] and dynamically allocated higher-order references [1]. Noteworthy recent applications of step-indexing involve a logical relation model of Rust [10] and a program logic based on session types [9].

The key idea of step-indexing is to use sequences of successive approximations to construct objects such as types, propositions, and functions, where the n-th approximation describes the object under the assumption that we only have n available computation steps to reason about it. To avoid tedious and low-level reasoning about step-indexed arithmetic, the *logical approach* to step-indexing [3, 8] employs the the *later* modality [12] to provide an abstraction.

The internal logic of the topos of trees [4] provides a full-fledged higher-order logic with dependent types, extended with a later modality on propositions ($\triangleright$) and a later modality on types ($\blacktriangleright$). This rich logic is a good candidate for defining program logics and logical relation models for complex programming languages internally.

Problem statement. Ideally, a logic should have neat and general rules for describing the interaction of the connectives. Unfortunately, the naive rules for commuting the later modality with quantifiers are unsound. Take the naive rule for commuting existentials and later:

$$\triangleright(\exists x : A.\, P) \dashv\vdash \exists x : A.\, \triangleright P$$

While the right-to-left direction follows from basic rules involving $\exists$ and $\triangleright$, the left-to-right direction does not hold because we quantify over different approximations of the domain A.

The current solution is to demand that the domain A is *total and inhabited* [4, 6]. This property can be expressed in the internal logic of the topos of trees as

$$\mathtt{TI}(A) := \forall y : \blacktriangleright A.\, \exists x : A.\, \mathtt{next}\, x = y.$$

Then we get the following inference rule:

$$\triangleright\text{-}\exists\text{-}\mathtt{TI}\quad \frac{\vdash \mathtt{TI}(A) \qquad \Gamma, x : A \vdash P : \mathtt{Prop}}{\Gamma \mid \triangleright(\exists x : A.\, P) \vdash \exists x : A.\, \triangleright P}$$

This rule is not ideal for two reasons. Firstly, it only works for total and inhabited types. Secondly, the assumption $\mathrm{TI}(A)$ has to be proved valid without any hypotheses. The second issue can be addressed by considering the following modified inference rule:

$$\triangleright\text{-}\exists\text{-}\mathtt{TI}\text{-}2\quad \frac{\Gamma, x : A \vdash P : \mathtt{Prop}}{\Gamma \mid \mathtt{TI}(A) \wedge \triangleright(\exists x : A.\, P) \vdash \exists x : A.\, \triangleright P}$$

This rule is also sound in the topos of trees, and it implies $\triangleright$-$\exists$-TI. However, the restriction on the domain of quantification remains.

Solution. Our proposed solution rests on the observation (first made by [4]) that, semantically speaking, the later modality $\triangleright : \mathtt{Prop} \to \mathtt{Prop}$ can be decomposed as $\triangleright = \mathtt{lift} \circ \mathtt{next}$, where $\mathtt{next} : \mathtt{Prop} \to \blacktriangleright\mathtt{Prop}$ and $\mathtt{lift} : \blacktriangleright\mathtt{Prop} \to \mathtt{Prop}$ (here, Prop is the type of propositions, *i.e.,* the subobject classifier in the topos of trees). Thus, we can study the interaction of the quantifiers with next and lift separately.

It follows from the homomorphism property of applicative functors that next commutes with the quantifiers, in the sense that we have equalities

$$\mathtt{next}\,(\exists x : A.\, Px) = \mathtt{next}\,\mathtt{ex} \circledast \mathtt{next}\,P \quad \text{and} \quad \mathtt{next}\,(\forall x : A.\, Px) = \mathtt{next}\,\mathtt{all} \circledast \mathtt{next}\,P,$$

where $\mathtt{ex}, \mathtt{all} : (A \to \mathtt{Prop}) \to \mathtt{Prop}$ are defined as:

$$\mathtt{ex}(P) = \exists x : A.\, Px \quad \text{and} \quad \mathtt{all}(P) = \forall x : A.\, Px.$$

Our contributions are the following two dual rules that relate lift with the quantifiers:

$$\mathtt{lift}\text{-}\exists\quad \frac{\Gamma \vdash Q : \blacktriangleright(A \to \mathtt{Prop})}{\Gamma \mid \mathtt{lift}\,(\mathtt{next}\,\mathtt{ex} \circledast Q) \dashv\vdash \exists y : \blacktriangleright A.\, \mathtt{lift}\,(Q \circledast y)}$$

$$\mathtt{lift}\text{-}\forall\quad \frac{\Gamma \vdash Q : \blacktriangleright(A \to \mathtt{Prop})}{\Gamma \mid \mathtt{lift}\,(\mathtt{next}\,\mathtt{all} \circledast Q) \dashv\vdash \forall y : \blacktriangleright A.\, \mathtt{lift}\,(Q \circledast y)}$$

It can be shown that lift-$\exists$ implies $\triangleright$-$\exists$-TI-2 and thus also $\triangleright$-$\exists$-TI. Hence, our new rules really are generalisations of the previously studied rules. This can be seen as an argument for the claim that lift is more primitive than $\triangleright$, and that step-indexed logics should focus on axiomatising the former.

Conclusion. We have found new and more general reasoning rules in the topos of trees for commuting the later modality with quantifiers. We proved the soundness of our new rules and showed that previously known rules can be derived from them. To ensure confidence in our results, we have formalised the topos of trees and its internal logic in the Rocq Prover [7].

It remains to be investigated how to apply our rules in practice. For instance, we would like to define a model of a step-indexed program logic (such as Iris [11]) in the internal logic of the topos of trees and see if our new rules are of use. Furthermore, one should investigate the interaction of `lift` with the other connectives in order to get a more complete axiomatisation of the internal logic.

We note that an operation similar to `lift`, called $\widehat{\triangleright}$, has already been studied in the context of guarded dependent type theory [5]. This operation is used to turn a code a for a type A to a code $\widehat{\triangleright}\, a$ for the type $\blacktriangleright A$. Under the Curry-Howard correspondence, propositions are expressed as types in dependent type theory, and thus, $\widehat{\triangleright}$ can be seen as a generalised version of `lift`. In such a setting, the rule `lift-∃` corresponds to the principle that the later type former preserves sigma types.

Finally, it would be worthwhile to investigate whether principles similar to `lift-∃` and `lift-∀` also hold in other models of step-indexing. In particular, it is conceivable that these rules are also valid in models where step-indexing is done over an ordinal larger than ω, such as Transfinite Iris [13].

Acknowledgments. We thank the anonymous reviewers for their helpful feedback.

References

[1] Amal J. Ahmed. *Semantics of types for mutable state.* PhD thesis, Princeton University, 2004.

[2] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001. doi:10.1145/504709.504712.

[3] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *POPL*, pages 109–122, 2007. doi:10.1145/1190216.1190235.

[4] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.*, 8(4), 2012. doi:10.2168/LMCS-8(4:1)2012.

[5] Ales Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In *FoSSaCS*, volume 9634 of *LNCS*, pages 20–35, 2016. doi:10.1007/978-3-662-49630-5_2.

[6] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types. *Log. Methods Comput. Sci.*, 12(3), 2016. doi:10.2168/LMCS-12(3:7)2016.

[7] The Rocq development team. The Rocq prover, 2025. URL `https://rocq-prover.org/`.

[8] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Log. Methods Comput. Sci.*, 7(2), 2011. doi:10.2168/LMCS-7(2:16)2011.

[9] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris 2.0: Asynchronous session-type based reasoning in separation logic. *Log. Methods Comput. Sci.*, 18 (2), 2022. doi:10.46298/LMCS-18(2:16)2022.

[10] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL): 66:1–66:34, 2018. doi:10.1145/3158154.

[11] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

[12] Hiroshi Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000. doi:10.1109/LICS.2000.855774.

[13] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. In *PLDI*, pages 80–95, 2021. doi:10.1145/3453483.3454031.

How (not) to prove typed type conversion transitive

Yann Leray

Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, F-44000 Nantes, France

Type conversion is the central part of implementations of type theories. It is theoretically decidable thanks to the normalisation of all well-typed terms. However, most if not all proof assistants do not in fact normalise all terms to test for convertibility, instead trying to check if they are syntactically equal as a shortcut. Tangentially, many languages used in proof assistants are not proved to be normalising, and some even allow extensions which explicitly prevent normalisation.

Without the assumption of normalisation, many properties required for the adequacy between such an algorithm and the intended specification become brittle and are left to be justified, with the most central of these being the transitivity of the relation defined by the algorithm. There currently exist proofs of the soundness of the algorithm for an untyped specification of PCUIC (the idealised type theory of Rocq) [SFL+25], and with a typed specification of Pure Type Systems (PTSs) [Ada06, SH12] with no extensionality (η) rules. We present the difficulties of replicating these proofs in the context of a typed specification, with extensionality rules, as well as the paths we explored and why we stopped exploring some.

Specification and algorithm. Let us describe the two approaches to type conversion that we want to link. On one end, the specification for convertibility of terms (denoted by $\equiv$) will include freely all reductions, η-rules and congruence rules, such that it will be an equivalence relation. It however doesn't satisfy any type constructor injectivity results out of the box, due to its transitivity rule muddying everything. On the other end, we will abstract the behaviour of an algorithm by the relation $\cong$, which does repeated head-reductions and (typed) head-η-expansions on both sides arbitrarily, and at any point when the head constructors are the same can it stop and resume the operation on the subterms instead (i.e. apply term constructor congruence rules).

With these definitions, inclusion of $\cong$ in $\equiv$ is immediate and, for the converse inclusion, most rules are also already present. Reflexivity and symmetry are both obtained by simple inductions, leaving only transitivity remaining. One should first notice that transitivity cannot be proven with a simple induction, for the same reason that confluence of λ-calculus cannot be derived by induction using simply its weak confluence property, because of an increasing reduction length issue: when you do deal with the case $\Gamma \vdash (\lambda(x : A_0), t_0)\ u_0 \equiv (\lambda(x : A_1), t_1)\ u_1 : T$ (by congruence) against $\Gamma \vdash (\lambda(x : A_1), t_1)\ u_1 \equiv t_2[x := u_2] : T$, you can derive $\Gamma \vdash t_0[x := u_0] \equiv t_1[x := u_1] : T$, but since it isn't a structurally smaller derivation than the one you start with, the induction cannot go through.

How to prove transitivity. In a context where we intend to prove normalisation and can afford to work only with normalising terms (e.g. [AÖV17, HJP23, ALM+24]), there is a simple proof of transitivity for $\cong$. It relies on the observation that only congruence rules are applicable to normal forms, which makes the relation transitive on such terms. It then only remains to show that all shortcuts that can be taken are admissible.

However, in our context where no proof of normalisation is accessible, we need another approach. Since our issue at hand is similar to what happens in λ-calculus, we can resort to the solution that is used there and was already extended to work for PCUIC and PTSs: parallel

reduction [Tak89] and the Hindley-Rosen technique [BKv98]. We will split the rules of $\cong$ into multiple relations which will be parallel reduction ($\Rrightarrow$, reduction rules and congruences), parallel η-expansion ($\Rrightarrow_\eta$, η-expansion and congruences) and syntactic equality ($=_s$, α-equivalence and type-annotation irrelevance). Our complete conversion relation will now be $(\Rrightarrow + \Rrightarrow_\eta)^*; =_s ; (\Lleftarrow +_\eta \Lleftarrow)^*$ (arbitrary reductions on both sides using both parallel reduction and parallel η-expansion, closed by syntactic equality).

The idea behind this split is that, starting from a concatenation of two conversions, we reorder the individual steps using some swapping lemmas and hope to reach a state where it has a the shape of a single conversion. Since we're working with parallel relations, we are looking for simulations between these relations ($R; R' \subseteq R'; R$ or sometimes $R; R' \subseteq R'; =_s; R$)

Compatibilities, constraints on type conversion. The proofs of the different swapping lemmas will require establishing some properties on terms, themselves further requiring some properties on the type conversion used in the parallel relations (abstract until now).

First, during the inversions that happen on the examined parallel relations, we need to relate the types of corresponding subterms, so we require that terms have principal types and that type conversion be reflexive and transitive. Second, since typing is based on the left term, our relations have to be stable under type and context change (for the equivalent context-level relations), which means that typing and type conversion has to be stable under such changes; here on top of that, type conversion also needs to be stable on both sides by all relations in both directions (e.g. both $\Rrightarrow$ and $\Lleftarrow$).

For the proof of the strong confluence (self-simulation) of $\Rrightarrow$, we observe that it has to be stable under substitution, which means in turn that both typing and type conversion have to be stable under substitution.

For the simulation between $\Rrightarrow$ and $\Rrightarrow_\eta$, we must disallow for a term t to have both $t : \mathbb{B}$ and $t : \Pi(x : A), B$, or for a term $t : \mathrm{Type}$ to have either $t : \mathbb{B}$ or $t : \Pi(x : A), B$. Since we also assume that terms have principal types, this translates into a requirement of non-confusion of type conversion for type formers, that is, no two among $\mathbb{B}$, $\Pi(x : A), B$ and Type should be convertible. Simultaneously, we require that all possible β- and ι-redexes may be reduced, i.e. any redex well typed as a term has its additional redex typing constraints satisfied. These typing constraints are all provable if we require the injectivity of type conversion for type formers (when two function types are convertible, then so are their domains and so are their codomains).

However, this whole proof sketch of transitivity is not structural, so we can't use a simple induction hypothesis to satisfy both our transitivity requirement and our stability under reduction requirements. This means we have to find an existing relation which satisfies all of our constraints, yet the list is long enough that our needed conversion is already what we want to find.

Side-stepping some constraints. In the talk we will report on ongoing investigations and here present the current status of our research for a conclusive proof. Let us first focus over the requirements we can fulfil. For type principality, two ways exist which don't require any previous results: using a bidirectional type system [Len21] or using over-annotated terms (where for instance an application $(f)\ a$ becomes $(f)\{(x : A : s_0), B : s_1\}\ a$ to add type annotations for all subterms). While the former approach looks simpler, the latter approach allows for context changes which do not influence principal types (thanks to the type annotation on variables), so we will focus on that one.

Working with principal types also helps reduce the requirements on type conversion we need: when one term reduces to another (through either $\Rightarrow$ or $\Rightarrow_\eta$), then the same is true of their principal types. On the one hand, this means that our relations now work on two types instead of one; we also need redexes to preserve principal types, so we may need to introduce explicit definitional casts in some reductions, and a specific reduction relation to erase them. On the other hand, this waives the requirement of both reflexivity and stability by $\Leftarrow$ and ${}_\eta\Leftarrow$ for type conversion. At this stage, this means that instantiating type conversion with the false relation would be sound (but obviously not complete).

The requirements for transitivity become located to a certain number of redexes (namely, η-expansions and cast erasing), where it can be assumed locally instead of globally, with hopes of transporting the assumption instead of reconstructing it when needed).

This leaves only the requirements of non-confusion and injectivity of type constructors, and stability under substitution and all parallel relations. Once more, neither $\equiv$ nor $\cong$ satisfy both properties out of the box, so we need another solution. The currently envisioned solution would be to devise an induction with a very strong predicate, such that it can satisfy all of our requirements at the same time. We first need to make sure that we won't need to construct any proofs of it, which is the reason why we tried to eliminate the requirement of reflexivity in the first place. A lot of work remains to be done to conclude the proof nonetheless.

References

[Ada06] Robin Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, March 2006.

[ALM+24] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2024, pages 230–245, New York, NY, USA, January 2024. Association for Computing Machinery.

[AÖV17] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL):23:1–23:29, December 2017.

[BKv98] Marc Bezem, Jan Willem Klop, and Vincent van Oostrom. Diagram Techniques for Confluence. *Information and Computation*, 141(2):172–204, March 1998.

[HJP23] Jason Z. S. Hu, Junyoung Jang, and Brigitte Pientka. Normalization by evaluation for modal dependent type theory. *Journal of Functional Programming*, 33:e7, January 2023.

[Len21] Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[SFL+25] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM*, 72(1):8:1–8:74, January 2025.

[SH12] Vincent Siles and Hugo Herbelin. Pure Type System conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, March 2012.

[Tak89] Masako Takahashi. Parallel reductions in λ-calculus. *Journal of Symbolic Computation*, 7(2):113–123, February 1989.

Arrow algebras

Benno van den Berg[1], Marcus Briët, and Umberto Tarantino[2]

[1] Institute for Logic, Language and Computation
Universiteit van Amsterdam
Postbus 90242, 1090 GE Amsterdam
The Netherlands
B.vandenBerg3@uva.nl
[2] IRIF
Université Paris Cité
Bâtiment Sophie Germain, Case courier 7014
8 Place Aurélie Nemour
75205 Paris Cedex 13
France
tarantino@irif.fr

Background. An *elementary topos* can be seen as a model of a version of the Calculus of Constructions with an impredicative universe of propositions, where any two elements of a proposition are definitionally equal. There is an extensive literature on topos theory (see, for example, [10, 8, 9]) and many properties of this type theory can be proved using its topos-theoretic semantics.

An important class of such toposes is the one obtained from *locales*. A locale L is a complete poset in which the following distributive law holds:

$$a \wedge \bigvee_{b \in B} b = \bigvee_{b \in B} a \wedge b,$$

when $a \in L$ and $B \subseteq L$. Every topological space gives rise to a locale by considering its poset of open subsets ordered by inclusion.

Whenever you have a locale, you can obtain a topos from it by taking the category of *sheaves* over the locale: the result is called a *localic topos*. This category of sheaves over the locale L is equivalent to a category that has a description in terms of logic. Indeed, there is an equivalent category of L-sets, which are sets with an L-valued equality relation on them, where this equality relation is required to be symmetric and transitive; the morphisms of L-sets are L-valued functional relations.

The latter category can be understood as the result of a two-step process. First, one builds a *tripos* out of the locale L and then one turns this tripos into a topos by the *tripos-to-topos construction* [6]. Importantly, there are triposes that do not arise from locales, for instance, the effective tripos, whose associated elementary topos is Hyland's effective topos, a non-localic (even non-Grothendieck) topos [5]. The effective topos and their subcategories are important as models of polymorphic type theories [4, 7].

Contribution. The aim of this talk is to introduce *arrow algebras* and explain the work of my former MSc students Marcus Briët and Umberto Tarantino [1, 15]. Arrow algebras are algebraic structures generalising locales. The point is that they still allow you to construct a tripos, an *arrow tripos*, and hence also an *arrow topos* by the tripos-to-topos construction.

These arrow toposes include the localic toposes, but also Hyland's effective topos. Indeed, many realizability toposes can be shown to be arrow toposes, because every *pca* (*partial combinatory algebra*) gives rise to an arrow algebra: this includes also "relative, ordered" pcas as in, for example, Zoethout's PhD thesis [16] (see also [3, 13]).

Crucially, Umberto Tarantino has developed a notion of morphism of arrow algebras which correspond to geometric morphisms between the associated triposes. This has allowed us to understand the following in purely arrow algebraic terms:

1. Every arrow morphism factors as a surjection followed by an inclusion, inducing the corresponding factorisation on the level of triposes and toposes.
2. Every subtripos of an arrow tripos coming from an arrow algebra L is induced by a *nucleus* on L. Given this nucleus, there is a simple construction of a new arrow algebra inducing the subtripos.

As a result, arrow algebras provide a flexible framework for constructing and studing new toposes.

Related work. Arrow algebras can be defined as follows:

Definition 0.1. *An* arrow algebra A *is a complete lattice* $(A, \preccurlyeq)$ *with an implication operator* $\to\colon A^{\mathrm{op}} \times A \to A$ *and a* separator $S \subseteq A$ *such that:*

1. *if* $a \in S$ *and* $a \preccurlyeq b$*, then* $b \in S$*.*
2. *if* $a, a \to b \in S$*, then also* $b \in S$*.*
3. S *contains the following* combinators*:*

$$\begin{aligned}
\mathbf{k} &:= \bigwedge_{a,b} a \to b \to a \\
\mathbf{s} &:= \bigwedge_{a,b,c} (a \to b \to c) \to (a \to b) \to (a \to c) \\
\mathbf{a} &:= \bigwedge_{a,(b_i)_{i\in I},(c_i)_{i\in I}} \big(\bigwedge_{i\in I} a \to b_i \to c_i\big) \to a \to \big(\bigwedge_{i\in I} b_i \to c_i\big)
\end{aligned}$$

Arrow algebras were directly inspired by Alexandre Miquel's work on *implicative algebras* [12]. His implicative algebras can be defined as arrow algebras in which the following axiom holds:

$$a \to \bigwedge_{b\in B} b = \bigwedge_{b\in B} a \to b.$$

We felt it was worthwhile to drop this axiom, because there are many natural examples of arrow algebras that do not satisfy it: this includes arrow algebras obtained from pcas and the arrow algebras obtained from nuclei. While it follows from Miquel's work that every arrow algebra is equivalent to an implicative algebra [1, 11], the equivalent implicative algebra is rather unwieldy and for doing concrete calculations, working with the original arrow algebra is easier.

Implicative algebras are closely related to *evidenced frames*, as in [2]. Another framework which subsumes both realizability and localic toposes is the work by Pieter Hofstra on *BCOs*

(*basic combinatory objects*) [3] (see also [14]). While every implicative algebra is a BCO, it is not clear how to obtain a BCO from an arrow algebra in such a way that the associated triposes are isomorphic. However, a more thorough investigation of these connections is left to future work.

References

[1] M. Briët and B. van den Berg. "Arrow algebras". arXiv:2308.14096. 2023.

[2] L. Cohen, E. Miquey, and R. Tate. "Evidenced Frames: A Unifying Framework Broadening Realizability Models". *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 2021, pp. 1–13.

[3] P.J.W. Hofstra. "All realizability is relative". *Math. Proc. Cambridge Philos. Soc.* 141.2 (2006), pp. 239–264.

[4] J.M.E. Hyland. "A small complete category". *Ann. Pure Appl. Logic* 40.2 (1988), pp. 135–165.

[5] J.M.E. Hyland. "The effective topos". *The L.E.J. Brouwer Centenary Symposium (Noordwijkerhout, 1981)*. Vol. 110. Stud. Logic Foundations Math. Amsterdam: North-Holland Publishing Co., 1982, pp. 165–216.

[6] J.M.E. Hyland, P.T. Johnstone, and A.M. Pitts. "Tripos theory". *Math. Proc. Cambridge Philos. Soc.* 88.2 (1980), pp. 205–231.

[7] J.M.E. Hyland, E.P. Robinson, and G. Rosolini. "The discrete objects in the effective topos". *Proc. London Math. Soc. (3)* 60.1 (1990), pp. 1–36.

[8] P.T. Johnstone. *Sketches of an elephant: a topos theory compendium. Volume 1*. Vol. 43. Oxf. Logic Guides. New York: Oxford University Press, 2002, pp. xxii+468+71.

[9] P.T. Johnstone. *Sketches of an elephant: a topos theory compendium. Volume 2*. Vol. 44. Oxf. Logic Guides. Oxford: Oxford University Press, 2002, pp. xxii+469–1089+71.

[10] S. Mac Lane and I. Moerdijk. *Sheaves in geometry and logic – A first introduction to topos theory*. Universitext. New York: Springer-Verlag, 1992, pp. xii+629.

[11] A. Miquel. "Implicative algebras II: Completeness w.r.t. Set-based triposes". arXiv:2011.09085. 2020.

[12] A. Miquel. "Implicative algebras: a new foundation for realizability and forcing". *Math. Struct. Comput. Sci.* 30.5 (2020), pp. 458–510.

[13] J. van Oosten. *Realizability: an introduction to its categorical side*. Vol. 152. Studies in Logic and the Foundations of Mathematics. Elsevier B. V., Amsterdam, 2008, pp. xvi+310.

[14] J. van Oosten and T. Zou. "Classical and relative realizability". *Theory Appl. Categ.* 31 (2016), Paper No. 22, 571–593.

[15] U. Tarantino. "A category of arrow algebras for modified realizability". *Theory and Applications of Categories* 44 (2025), pp. 132–180.

[16] J. Zoethout. "Computability models and realizability toposes". PhD thesis. Utrecht University, 2022.

Weak Equality Reflection in MLTT with Propositional Truncation

Felix Bradley and Zhaohui Luo

Royal Holloway, University of London, Egham, U.K.
felix.bradley@rhul.ac.uk
zhaohui.luo@hotmail.co.uk

Introduction In type theory, there are two primary kinds of equality: judgemental equality, and propositional equality. The former is definitional and takes the form of equality judgements $\Gamma \vdash a = b : A$, where as the latter inhabits the theory's internal logic wherein you can have proof objects to associate an identity or equality between two objects, e.g. of the form $p : \mathrm{Id}_A(a, b)$. Equality reflection is a statement about the relationship between these two notions of equality: strong equality reflection allows for judgemental equality to be derived from propositional equality within a system, whereas weak equality reflection simply asks whether these two different notions of equality coincide. A theory with strong equality reflection, such as an extensional type theory, may have an inference rule such as

$$\frac{\Gamma \vdash p : \mathrm{Eq}_A(x, y)}{\Gamma \vdash x = y : A}$$

whereas a system with weak equality reflection is defined to be one such that at least one of the following rules[1]

$$\frac{\langle\rangle \vdash p : \mathrm{Id}_A(x, y)}{\langle\rangle \vdash x = y : A} \qquad \frac{\langle\rangle \vdash p : x =_A y}{\langle\rangle \vdash x = y : A}$$

is admissible within the system for the empty context, but not necessarily derivable. Weak equality reflection holds in systems such as MLTT [6, 8] and UTT [3], but fails to hold in systems such as traditional homotopy type theory[2] [2].

Some developed applications of type theory, such as program specification/analysis and modern type theory semantics for natural language [9, 1, 5], have been developed with the use of weak equality reflection in mind. For example, weak equality reflection in program specification is critically used to define what is expected from definitional and computational equality. Some applications allow for various system to serve as a sufficient foundation - however, MLTT presents issues for some of applications in natural language semantics due to the counting problem [11]. This work analyses a proposed solution in $\mathrm{MLTT}_{\mathrm{h}}$ - the type theory MLTT extended with propositional truncation - and examines weak equality reflection within this system.

The Counting Problem For a sentence such as 'the number of black cats in the garden is one', one may provide type-theoretic semantics for the adjectival modification as a dependent pair type, as first proposed and studied by Mönnich [7] and Sundholm [10]. as

[1] Id is the identity type as found within MLTT [8], and $=_A$ is Leibniz equality as found within Luo's UTT [3].

[2] Consider the mere proposition $\Sigma(x : S^1).\,\mathrm{Id}(S^1, \mathrm{base}, x)$, where S^1 is the higher inductive type of the circle as typically defined.

$|\Sigma(x : \text{Cat}).\mathit{black}(x)| = 1$. However, as theories such as MLTT use a propositions-as-types logic and lack proof irrelevance, it may be the case that there exist multiple distinct proofs that a cat is black, and so the above semantics would be incorrect. As such, MLTT alone is not adequate as a foundational system for these purposes - but MLTT extended with the proper features can be. It was proposed that MLTT_h - the extension of MLTT with propositional truncation [2] - could serve as an adequate system for formalising modern type theory semantics for natural language [4].

Propositional Truncation MLTT_h is defined[3] as MLTT extended with propositional truncation - a type-level operator $\|-\|$ such that

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash |a| : \|A\|} \qquad \frac{\Gamma \vdash A\,\text{type}}{\Gamma \vdash p : \text{isProp}(\|A\|)} \qquad \frac{\Gamma \vdash \text{isProp}(B)\,\text{true} \quad \Gamma \vdash f : A \to B}{\Gamma \vdash \kappa_A(f) : \|A\| \to B}$$

where the elimination operator κ_A satisfies the definitional equality $\kappa_A(f, |a|) = f(a)$. Importantly, this introduces a kind of higher inductive type to the theory by mandating that any two objects of $\|A\|$ must be propositionally equal.

This allows us to define the traditional logical operators for this system's internal logic as follows:

- $\mathit{true} = \mathbf{1}$
- $\mathit{false} = \mathbf{1}$
- $P \mathbin{\dot\wedge} Q = P \times Q$
- $P \mathbin{\dot\vee} Q = \|P + Q\|$
- $P \mathbin{\dot\supset} Q = P \to Q$
- $\dot\neg P = P \to \mathbf{0}$
- $\dot\forall(x : P).Q = \Pi(x : P).Q$
- $\dot\exists(x : P).Q = \|\Sigma(x : P).Q\|$

Weak Equality Reflection This new internal logic replaces MLTT's propositions-as-types logic with a system built on proof irrelevance, allowing for MLTT_h to serve as a foundational system for e.g. natural language semantics. However, it is easy to show that this system no longer has weak equality reflection. For two terms $a : A$ and $b : B$, we can consider the example of $\|A + B\|$; we can conclude that $|\text{inl}(a)|$ and $|\text{inr}(b)|$ are propositionally equal as objects of a mere proposition, yet judgementally distinct due to their constructors.

However, because of how MLTT_h is constructed, there is a subtheory of it that resembles MLTT. Weak equality reflection holds for MLTT, so this subtheory of MLTT_h should preserve weak equality reflection, even if it does not hold for MLTT_h as a whole. In this case, MLTT_h could still be suitable for the applications discussed earlier.

One method for proving this being considered is through showing that MLTT_h is a conservative extension of MLTT. In particular, one would want to show that, for every context Γ and type A obtained in MLTT, if there exists some term a in MLTT_h such that $\Gamma \vdash a : A$, then there exists some term a' in MLTT such that $\Gamma \vdash a' : A$. If one is able to prove this then, because weak equality reflection holds for MLTT, it follows that the MLTT-like subtheory of MLTT_h must also preserve weak equality reflection.

Conclusion Our work on extending MLTT with propositional truncation and analysing its properties shows that MLTT_h is able to function as a foundational language for applications

[3] Some other works by the second author also refer to this system as MLTT *with h-logic* [4].

such as in natural language semantics, even if the theory as a whole does not necessarily preserve weak equality reflection. It also shows that some subsets of homotopy type theory are not able to fully preserve weak equality reflection, which suggests that it may not be possible for a notion of higher inductive types and weak equality reflection to exist within the same system. Our currently in-progress work aims to refine these results, and to further explore expanding MLTT_{h} with features which are useful but may not have been fully developed yet for homotopy type theory in general, such as with coercive subtyping.

References

[1] Stergios Chatzikyriakidis and Zhaohui Luo. *Formal Semantics in Modern Type Theories*. Wiley/ISTE, 2020.

[2] HoTT. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, The Univalent Foundations Program, Institute for Advanced Study, 2013.

[3] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, London, March 1994.

[4] Zhaohui Luo. Proof irrelevance in type-theoretical semantics. *Logic and Algorithms in Computational Linguistics 2018 (LACompLing2018), Studies in Computational Intelligence 860*, pages 1–15, 2019.

[5] Zhaohui Luo. *Modern Type Theories: Their Development and Applications*. Tsinghua University Press, 2024. (In Chinese).

[6] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. *Studies in Logic and the Foundations of Math*, 82, 1975.

[7] U. Mönnich. *Untersuchungen zu einer konstruktiven Semantik fur ein Fragment des Englischen*. Habilitation. University of Tübingen, 1985.

[8] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. International Series of Monographs in Computer Science. Oxford University Press, London, June 1990.

[9] Aarne Ranta. *Type-Theoretical Grammar*. Oxford University Press, Oxford, 1994.

[10] Göran Sundholm. Proof theory and meaning. In *Handbook of philosophical logic*, pages 471–506. Springer, 1986.

[11] Göran Sundholm. Constructive generalized quantifiers. *Synthese*, 79(1):1–12, 1989.

Towards Quotient Inductive Types in Observational Type Theory

Thiago Felicissimo and Nicolas Tabareau

INRIA, France

Quotient Inductive Types (QITs) are a family of inductive types supporting not only the declaration of generators but also equations. A well-known example of QIT is the following type of finite multisets, whose elements are basically lists considered up to permutation.

$$\begin{aligned}&\text{Inductive } \mathsf{MSet}\ (A : \mathrm{Type}) : \mathrm{Type} :=\\ &\quad |\ [] : \mathsf{MSet}\ A\\ &\quad |\ \_ :: \_\ (x : A)(m : \mathsf{MSet}\ A) : \mathsf{MSet}\ A\\ &\quad |\ \mathsf{MSet}_{=}\ (x\ y : A)(m : \mathsf{MSet}\ A) : (x :: y :: m) = (y :: x :: m)\end{aligned}$$

QITs are a special case of Higher Inductive Types (HITs) in which the defined type is implicitly declared to be an hSet. Such types has been gaining much interest in the last years [Uni13]. Notably, Altenkirch and Kaposi have proposed the use of Quotient Inductive-Inductive Types (QIITs), an extension of QITs with induction-induction, to formalize the metatheory of type theory in an intrinsic fashion [AK16].

QI(I)Ts have been investigated in works by Kovács *et al.* [KKA19, KK20] and Fiore *et al.* [FPS22], in both cases by proposing universal types which can be used for encoding other QI(I)Ts. However, they only focus on the case of extensional type theory (ETT), which is hard to implement in proof assistants due to its undecidable conversion. On the other hand, it is also known that the intensional type theories of proof assistants such as Rocq, Lean and Agda do not behave well with QITs. Indeed, the equality axioms that are added when declaring a QIT break canonicity, given that J does not reduce when eliminating them. Thankfully, we can instead shift the focus to Observational Type Theory (OTT) [AMS07, PT22], in which the eliminator for equality does not inspect the equality proof but instead computes using its endpoints. Because of this, (consistent) equality axioms can be safely added to the theory without breaking canonicity.

This work We report a work in progress metatheoretic study of OTT with QITs. In order to achieve this, many directions seem possible. Pujet and Tabareau have recently justified extensions of OTT with both quotient types Q [PT22] and inductive types [PLT25][1], therefore it can seem tempting to justify QITs by encoding them as regular inductive types quotiented by Q. Unfortunately, this encoding does not yield an eliminator with the proper definitional equalities, and for infinitary QITs this construction does not seem even possible [LS20]. Another approach would be to directly extend OTT with an inductive scheme for QITs and try to prove its metatheory, yet inductive schemes are hard to study in a formal manner due to the proliferation of indexes and vector notations. Therefore, we instead take a similar approach as the previous work on QITs, and extend OTT with a universal type which can be used to encode (non-indexed) QITs, by proposing an adaptation of Fiore *et al.*'s QW types.

[1] A version of OTT with Q and inductive types was actually already implemented in the Epigram 2 proof assistant, though, to the best of our knowledge, their metatheory had not yet been studied.

A universal QIT We now present a finitary version of our universal QIT, an infinitary version can also be found in the formalization.[2] Similarly to QW types, our definition goes in two steps: we start by defining a (unquotiented) type $\overline{\mathsf{Tm}}$, required to formulate the equations used for the quotient later. To do this, let signatures Sig be the record type with a field $\mathsf{C} : \mathrm{Type}$ to represent the type of constructors, and a field $\mathsf{arity} : \mathsf{C} \to \mathsf{Nat}$ which maps each constructor to its number of recursive arguments. We then define $\overline{\mathsf{Tm}}$ in the following way. Here, the parameter $\Gamma : \mathrm{Type}$ should be seen as a context which adds a new inhabitant $\mathsf{var}\ x$ for each $x : \Gamma$.

$$\begin{aligned}
&\mathrm{Inductive}\ \overline{\mathsf{Tm}}\ (\Sigma : \mathsf{Sig})(\Gamma : \mathrm{Type}) : \mathrm{Type} :=\\
&\quad |\ \mathsf{var}\ (x : \Gamma) : \overline{\mathsf{Tm}}\ \Sigma\ \Gamma\\
&\quad |\ \mathsf{sym}\ (c : \Sigma.\mathsf{C})\ (\mathbf{t} : \mathsf{Vec}\ (\overline{\mathsf{Tm}}\ \Sigma\ \Gamma)\ (\Sigma.\mathsf{arity}\ c)) : \overline{\mathsf{Tm}}\ \Sigma\ \Gamma
\end{aligned}$$

Using the above type, we define equational theories $\mathsf{EqTh}\ \Sigma$ to be the record type with a field $\mathsf{E} : \mathrm{Type}$ for the type of equations, a field $\mathsf{Ctx} : \mathsf{E} \to \mathrm{Type}$ mapping each equation to a type representing its context, and fields $\mathsf{lhs}, \mathsf{rhs} : (e : \mathsf{E}) \to \overline{\mathsf{Tm}}\ \Sigma\ (\mathsf{Ctx}\ e)$ mapping each equation to its left- and right-hand sides. We then define our universal QIT in the following manner. In the type of eq, we write $\_\langle\_\rangle : \overline{\mathsf{Tm}}\ \Sigma\ \Gamma \to (\Gamma \to \mathsf{Tm}\ \Sigma\ \mathcal{E}) \to \mathsf{Tm}\ \Sigma\ \mathcal{E}$ for the "substitution" function defined by $(\mathsf{sym}\ c\ [t_1 \dots t_k])\langle\gamma\rangle := \mathsf{sym}\ c\ [t_1\langle\gamma\rangle \dots t_k\langle\gamma\rangle]$ and $(\mathsf{var}\ x)\langle\gamma\rangle := \gamma\ x$.

$$\begin{aligned}
&\mathrm{Inductive}\ \mathsf{Tm}\ (\Sigma : \mathsf{Sig})\ (\mathcal{E} : \mathsf{EqTh}\ \Sigma) : \mathrm{Type} :=\\
&\quad |\ \mathsf{sym}\ (c : \Sigma.\mathsf{C})\ (\mathbf{t} : \mathsf{Vec}\ (\mathsf{Tm}\ \Sigma\ \mathcal{E})\ (\Sigma.\mathsf{arity}\ c)) : \mathsf{Tm}\ \Sigma\ \mathcal{E}\\
&\quad |\ \mathsf{eq}\ (e : \mathcal{E}.\mathsf{E})\ (\gamma : \mathcal{E}.\mathsf{Ctx}\ e \to \mathsf{Tm}\ \Sigma\ \mathcal{E}) : (\mathcal{E}.\mathsf{lhs}\ e)\langle\gamma\rangle = (\mathcal{E}.\mathsf{rhs}\ e)\langle\gamma\rangle
\end{aligned}$$

The above type can be used to define other (non-indexed) QITs. For instance, in the formalization we use it to define the type of finite multisets as well as the untyped SK calculus. We also provide the example of countably-branching trees, using the infinitary version of Tm.

Compared with QW types, our type is designed to employ a first-order representation of recursive arguments (an approach notably promoted by Dagand and McBride [Dag13]), by using vectors instead of functions. This aspect seems essential if we want encodings of QITs to satisfy canonicity: for instance, the encoding of the boolean type using QW types would have infinitely many normal inhabitants in the empty context.[3] Moreover, unlike with W and QW types, encodings using our type do not require functional extensionality or fancy tricks like [Hug21]: they work directly in intensional type theory, as witnessed by our Agda formalization.

The plan We plan to provide a metatheoretic justification of OTT with QITs using the above universal type, in three steps. (1) First, we will give an inductive scheme for non-indexed QITs and prove that all types can be encoded using our universal QIT. Importantly, we aim at obtaining eliminators that compute definitionally, and types that satisfy canonicity (eg., the encoding of the boolean type should only have two normal forms in the empty context). (2) Then, the second step will be to prove normalization of OTT extended with our universal QIT. While a pen-and-paper proof seems within reach, we expect that a formalization would require some important effort. Indeed, to the best of our knowledge, even the simpler W types lack a formalized normalization proof. (3) Finally, because in OTT consistency is not a consequence of normalization, we will adapt the set-theoretic model of Pujet and Tabareau to handle our universal QIT. Putting (2) and (3) together, we will then conclude canonicity of the type theory, from which we can also deduce canonicity of the encoded QITs.

[2] https://github.com/thiagofelicissimo/universal-QITs

[3] In the case of ITT, this can be fixed using Jasper Hugunin's trick [Hug21], however this is not sufficient in the case of OTT because equality does not satisfy canonicity in this setting.

Based on the above metatheoretic justification, we plan in the future to add QITs to the Rocq implementation of OTT [PLT25, Section 7]. We would of course also like to consider more complex classes of types, such as indexed QITs and QIITs. A promising direction would be to consider the universal type of Kovács *et al.* [KKA19]. However, because of the size of its definition, we are worried that a normalization proof would be harder to achieve. A different approach would be to justify this type using more basic type formers [Kov23, Section 4.6], however we are not sure this can be made to yield an eliminator with the proper definitional equalities.

Finally, an alternative approach to the one we propose here would be to instead consider Cubical Type Theory [CCHM18], which also behaves well with QITs, and whose implementation in Cubical Agda supports HITs in general [VMA19]. Although canonicity and normalization are known to hold in the presence of some specific HITs [Ste22, Hub19], to the best of our knowledge a full metatheoretic treatment of QITs in Cubical Type Theory is also lacking. A possible way to address this could be to use van der Weide and Geuvers's construction of finitary QITs from quotient types and proposition truncation in HoTT [vG19], yet the elimination principles they obtain only satisfy the expected equalities propositionally. Alternatively, one could try to adapt the strategy we propose here, by first constructing QITs from Tm and then studying the metatheory of Cubical Type Theory with Tm. Yet here we chose to focus only on the case of OTT, because we want a type theory for set-level mathematics.

References

[AK16] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *SIGPLAN Not.*, 51(1):18–29, jan 2016.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, 2007.

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Dag13] Pierre-Évariste Dagand. *A cosmology of datatypes : reusability and dependent types*. PhD thesis, University of Strathclyde, Glasgow, UK, 2013.

[FPS22] Marcelo P Fiore, Andrew M Pitts, and SC Steenkamp. Quotients, inductive types, and quotient inductive types. *Logical Methods in Computer Science*, 18, 2022.

[Hub19] Simon Huber. Canonicity for cubical type theory. *J. Autom. Reason.*, 63(2):173–210, August 2019.

[Hug21] Jasper Hugunin. Why Not W? In *26th International Conference on Types for Proofs and Programs (TYPES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[KK20] András Kovács and Ambrus Kaposi. Large and infinitary quotient inductive-inductive types. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, page 648–661, New York, NY, USA, 2020. Association for Computing Machinery.

[KKA19] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[Kov23] András Kovács. Type-theoretic signatures for algebraic theories and inductive types, 2023.

[LS20] Peter LeFanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):159–208, 2020.

[PLT25] Loïc Pujet, Yann Leray, and Nicolas Tabareau. Observational Equality Meets CIC. *ACM Trans. Program. Lang. Syst.*, February 2025. Just Accepted.

[PT22] Loïc Pujet and Nicolas Tabareau. Observational equality: now for good. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–27, 2022.

[Ste22] Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, 2022.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[vG19] Niels van der Weide and Herman Geuvers. The construction of set-truncated higher inductive types. *Electronic Notes in Theoretical Computer Science*, 347:261–280, 2019. Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics.

[VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

Matching (Co)patterns with Cyclic Proofs

Lide Grotenhuis, University of Amsterdam
Daniël Otten, University of Amsterdam

Overview. We investigate connections between cyclic proof theory and recursive functions defined by (co)pattern matching. Cyclic proof systems replace (co)induction rules with sound forms of circular reasoning. For example, by adding a cycle between the green nodes we get a cyclic proof:

$$\dfrac{\dfrac{}{0+0=0}\,{+_0} \qquad \dfrac{\dfrac{}{0+\mathsf{suc}\,x'=\mathsf{suc}(0+x')}\,{+_{\mathsf{suc}}} \quad \dfrac{0+x'=x'}{\mathsf{suc}(0+x')=\mathsf{suc}\,x'}\,\mathrm{cong}_{\mathsf{suc}}}{0+\mathsf{suc}\,x'=\mathsf{suc}\,x'}\,\mathrm{trans}}{0+x=x}\,\mathrm{case}_x,$$

This proof is sound because the variable x is decreased to its predecessor x' before we cycle back, and so it represents a proof by infinite descent. The advantage of these systems lies in proof search: to apply (co)induction we need to guess the right (co)induction hypothesis, whereas with cycles we can start generating the proof until our current goal matches one that we have seen before. Under the Curry-Howard correspondence, such cycles correspond to recursive function calls, while the soundness condition ensure that the function always terminates:

cyclic proof	recursive function
fixpoint formula	(co)inductive type
cycle	recursive function call
soundness condition	termination checking

In this way, recursive functions defined with dependent (co)pattern matching can be seen as a proof-relevant and dependent generalisation of cyclic proofs. In addition, there is a correspondence between type-based termination checking, in the form of sized-types, and ordinal approximations of fixpoint formulas. Our goal is to explain these correspondences and to use these connections to extend type theoretic conservativity results.

Proof Assistants. Proof assistants like `Agda`, `Dedukti`, `Rocq`, and `Lean` allow the user to define functions using recursive calls. Which functions are accepted depends on a unification algorithm, and this determines in particular whether we can prove axiom K. For both cases — with and without K — we have conservativity results: we can already implement these functions using the primitive rules (turn these cyclic proofs into (co)inductive proofs) [GMM06, CDP16, Thi20]. Such a reduction is needed to show that any function accepted by the proof assistant has an interpretation in any model of the type theory. However, these conservativity results only cover 'simple cycles': there is one inductive input that is decreased in every recursive call, or one coinductive output that is productive before every recursive call. Some proof assistants, like `Agda` and `Dedukti`, allow more complex interleaving of recursive calls. To implement these functions using primitive (co)induction rules, we need to apply induction and coinduction multiple times, and often in a specific order. For example, for the Ackermann function we need

a lexicographical order on inputs:

$$\begin{array}{l}\mathsf{ack} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, \\ \mathsf{ack}\, m\, n \coloneqq \mathsf{case}\, m \begin{cases} 0 \mapsto \mathsf{suc}\, n, \\ \mathsf{suc}\, m' \mapsto \mathsf{case}\, n \begin{cases} 0 \mapsto \mathsf{ack}\, m'\, 1, \\ \mathsf{suc}\, n' \mapsto \mathsf{ack}\, m'\, (\mathsf{ack}\, (\mathsf{suc}\, m')\, n'). \end{cases} \end{cases}\end{array}$$

While the following example mixes induction and coinduction:

$$\begin{array}{l}\mathsf{complog} : \mathbb{N} \to \mathbb{N} \to \mathsf{Stream}\, \mathbb{N}, \\ \mathsf{complog}\, t\, r \coloneqq \mathsf{case}\, t \begin{cases} 0 \mapsto \mathsf{record} \begin{cases} \mathsf{head} \mapsto r, \\ \mathsf{tail} \mapsto \mathsf{complog}\, r\, r, \end{cases} \\ \mathsf{suc}\, t' \mapsto \mathsf{complog}\, t'\, (r!). \end{cases}\end{array}$$

And this one uses induction on both inputs simultaneously:

$$\begin{array}{l}\mathsf{aggr} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, \\ \mathsf{aggr}\, m\, n \coloneqq \mathsf{case}\, m \begin{cases} 0 \mapsto 0, \\ \mathsf{suc}\, m' \mapsto \mathsf{case}\, n \begin{cases} 0 \mapsto \mathsf{suc}\, 0, \\ \mathsf{suc}\, n' \mapsto \mathsf{aggr}\, m'\, m' + \mathsf{aggr}\, n'\, n'. \end{cases} \end{cases}\end{array}$$

In addition, there are more complicated examples using dependent types and mutually recursive functions. The general soundness condition is the following: for any infinite sequence of function calls that might occur, there should eventually be an input/output that we can track, where progress is made infinitely often. This can either be an inductive input that is decreased infinitely often, or a coinductive output that is productive infinitely often. This is known as the size-change termination principle [LJBA01, Wah07], which can be implemented both with and without sized-types. Although this problem refers to infinite sequences, it can be shown to be PSPACE-complete, and can be solved using ω-automata or call graphs. This principle ensures termination, but it is no longer clear how the accepted functions can be implemented using primitive (co)induction rules.

Cyclic proofs. A similar problem has been studied in the cyclic proof theory literature [SD03b, CD23]. Here the setting is non-dependent and proof-irrelevant, but we see the same complex interleaving of cycles. Instead of tracing inputs and outputs for functions, we are tracing formulas through an infinite sequent calculus derivation, which is obtained by unfolding the cyclic derivation. We then have a similar PSPACE-complete soundness condition: for every infinite branch of the derivation, we can eventually trace a fixpoint formula that makes progress infinitely often [SD03a]. By a well-known strategy from cyclic proof theory, one can turn this global soundness condition into a local one by adding so-called *annotations* to formulas [Sti14, Jun10, DKMV23, LW24]. These annotations are based on Safra's determinisation construction [Saf88] for ω-automata.

Outlook. Inspired by the proof-theoretic picture, we hope to extend the current type theoretic conservativity results by allowing more complex interleaving function calls. Our hope is that by annotating the call graph, we can move from a global to a local soundness condition. These annotations then inform us on the order of (co)induction, which in turn should allow us to turn cyclic proofs into inductive proofs, following a similar approach to Sprenger and Dam [SD03b].

References

[CD23] Gianluca Curzi and Anupam Das. Computational expressivity of (circular) proofs with fixed points. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2023.

[CDP16] Jesper Cockx, Dominique Devriese, and Frank Piessens. Eliminating dependent pattern matching without k. *Journal of functional programming*, 26:e16, 2016.

[DKMV23] Maurice Dekker, Johannes Kloibhofer, Johannes Marti, and Yde Venema. Proof systems for the modal-calculus obtained by determinizing automata. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 242–259. Springer, 2023.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. *Eliminating Dependent Pattern Matching*, pages 521–540. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Jun10] Natthapong Jungteerapanich. *Tableau systems for the modal μ-calculus*. PhD thesis, University of Edinburgh, 2010.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *SIGPLAN Not.*, 36(3):81–92, January 2001.

[LW24] Graham E. Leigh and Dominik Wehr. From gtc to image 1: Generating reset proof systems from cyclic proof systems. *Annals of Pure and Applied Logic*, 175(10):103485, 2024.

[Saf88] S. Safra. On the complexity of omega -automata. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 319–327, 1988.

[SD03a] Christoph Sprenger and Mads Dam. On global induction mechanisms in a μ-calculus with explicit approximations. *RAIRO-Theoretical Informatics and Applications*, 37(4):365–391, 2003.

[SD03b] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, pages 425–440, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Sti14] Colin Stirling. A tableau proof system with names for modal mu-calculus. In *HOWARD-60*, 2014.

[Thi20] David Thibodeau. *An Intensional Type Theory of Coinduction Using Copatterns*. PhD thesis, McGill University Montréal, QC, Canada, 2020.

[Wah07] David Wahlstedt. *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion*. Chalmers Tekniska Hogskola (Sweden), 2007.

Towards a Computational Quantum Logic: An Overview of an Ongoing Research Program*

Alejandro Díaz-Caro

[1] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
[2] Universidad Nacional de Quilmes, Bernal, Buenos Aires, Argentina

Abstract

This extended abstract provides an overview of a long-term collaboration aimed at extending the Curry-Howard-Lambek correspondence to the realm of quantum computation. I will introduce the Lambda-S calculus, a (partial) dual to intuitionistic linear logic whose proof terms serve as a foundation for a quantum programming language. Additionally, I will discuss the $\mathcal{L}^{\mathcal{S}}$ calculus, a proof language for intuitionistic linear logic, which also enables the construction of quantum programming languages. These frameworks offer a logical and computational foundation for reasoning about quantum programs and provide a glimpse into the structure of a potential quantum logic as the dual of linear logic.

Introduction. Quantum logic is a formal system inspired by the structure of quantum theory, originally developed by Garrett Birkhoff and John von Neumann [BvN36]. Unlike classical logic, which is based on Boolean algebra, quantum logic weakens the distributive law, leading to an orthocomplemented lattice structure. This formulation aligns with the mathematical properties of quantum mechanics, where propositions correspond to projections on a Hilbert space. However, while quantum logic has been explored as a foundational system for reasoning about quantum mechanics, its connection to computation has been less explored. Indeed, the connection between intuitionistic logic, typed lambda calculus, and Cartesian closed categories has been a fruitful area of research, with the Curry-Howard-Lambek correspondence [SU06, Cro93] providing a deep connection between these areas. If we are to extend this correspondence to quantum computation, we need to start from a logical foundation that captures the structure of quantum computation. In this extended abstract, I will provide an overview of a research program aimed at developing a *computational* quantum logic, which will serve as a foundation for quantum programming languages. This program is based on two main frameworks: the Lambda-S calculus, which is an extension of the lambda calculus to quantum computing, and the $\mathcal{L}^{\mathcal{S}}$ calculus, which is a proof language for intuitionistic linear logic whose proof terms can be used to construct quantum programs. The idea was to start from computing to logic (the Lambda-S calculus), and then from logic to computing (the $\mathcal{L}^{\mathcal{S}}$ calculus), in order to meet in the middle.

The Lambda-S Calculus. From Computing to Logic. Lambda-S [DCDR19] and Lambda-S_1 [DCGMV19] are quantum lambda calculi designed to handle quantum superpositions and control while preserving computational properties such as strong normalization and subject reduction. Both calculi extend the lambda calculus with algebraic linearity and type-based constraints to enforce quantum mechanics principles, particularly the no-cloning theorem.

Lambda-S is an extension of simply typed lambda calculus with linear combinations of terms, it incorporates a type constructor $S(A)$ where a simple type A denotes a set of basis vectors

*A longer version of this abstract will appear at [DC25].

(terms) and $S(A)$ its span. This system ensures that terms in A are duplicable, whereas terms in $S(A)$ are not, reflecting the impossibility of cloning unknown quantum states. Lambda-S has been given a categorical semantics through an adjunction between Cartesian and additive symmetric monoidal categories, where S is a functor transforming sets into vector spaces, and its adjoint forgets the vectorial structure [DCM23, DCM20].

Lambda-S_1 extends Lambda-S by enforcing norm-preserving constraints on superpositions, making it a stricter model suitable for representing unitary transformations explicitly. It has been obtained via a realizability model in [DCGMV19]. The categorical semantics of Lambda-S_1 [DCM22] is structured around adjunctions but diverges from Lambda-S by ensuring that all terms maintain a unitary norm, addressing the long-standing issue of preserving quantum unitarity in quantum control lambda calculi.

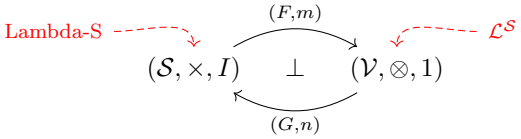
Both calculi serve as foundations for quantum programming languages and categorical models of quantum computation, bridging classical and quantum computational paradigms through rigorous mathematical structures.

One of the most notable results of this *side* of the research program, is the fact that we proved the model of Lambda-S to be a (partial) dual of known models for intuitionistic linear logic (ILL). Partial, because it favors a computational basis, but some preliminary results suggest that it can be extended to a proper dual [Mon25]. Usually, ILL is interpreted in a monoidal closed category, using an adjunction with a Cartesian closed category to interpret duplicable data. Lambda-S, on the other hand, is interpreted in a Cartesian closed category, using the same adjunction, but the other way around, to interpret non-duplicable data. This duality is a strong indication that the structure of quantum computation is the dual of linear logic, and that the Curry-Howard-Lambek correspondence can be extended to quantum computation, with this linear logic dual as the logical side and Lambda-S as the computational side.

The $\mathcal{L}^{\mathcal{S}}$ Calculus. From Logic to Computing. The $\mathcal{L}^{\mathcal{S}}$ calculus extends intuitionistic multiplicative additive linear logic (IMALL) with algebraic structures, incorporating sum and scalar multiplication within proof terms. It builds on the $\odot$-calculus [DCD23], which introduced an algebraic connective for quantum superpositions, and refines this idea within a fully linear framework [DCD24]. In a recent draft, we replace the $\odot$ connective with an alternative rule for disjunction introduction, enabling a structured representation of the quantum measurement [DCD25]. Its categorical semantics formalizes this approach, aligning it with monoidal structures that preserve linearity [DCM24]. Moreover, the calculus has been extended [DCDIM24] with !, that is, from IMALL to ILL, and polymorphism, ensuring expressive power suitable for quantum programming languages.

The $\mathcal{L}^{\mathcal{S}}$ calculus provides a logical foundation for quantum programming languages, enabling the construction of quantum circuits and algorithms through proof terms. Its algebraic structure aligns with quantum mechanics principles, allowing for a direct representation of quantum superpositions and measurements. The calculus serves as a bridge between linear logic and quantum computation, providing a formal framework for reasoning about quantum programs.

The fact that both languages, Lambda-S and $\mathcal{L}^{\mathcal{S}}$, use the same adjunction



where $\mathcal{S}$ is a Cartesian closed category, $\mathcal{V}$ is a monoidal closed category, but with Lambda-S being interpreted in $\mathcal{S}$, using the monad GF to interpret the non-duplicable terms, and $\mathcal{L}^{\mathcal{S}}$

being interpreted in $\mathcal{V}$, using the comonad FG to interpret the duplicable terms, is a strong indication that the structure of quantum computation can be seen as the dual of linear logic, and that the Curry-Howard-Lambek correspondence can be extended to quantum computation following this path.

Several open challenges remain. On the one hand, while preliminary results suggest a duality between quantum computation and linear logic, a complete characterisation of this duality, both syntactically and categorically, is still an open question. On the other hand, extending the Lambda-S and $\mathcal{L}^{\mathcal{S}}$ calculi to fully capture quantum measurement processes, particularly in the presence of multiple bases and dependent types, demands further development. Additionally, establishing a formal correspondence between the algebraic structures of the calculi and graphical languages such as the ZX-calculus, and exploiting this link for program verification and optimisation, represents another promising but challenging direction.

This ongoing research program is advancing steadily, and the promising results so far suggest that computational quantum logic is a viable and interesting direction.

Acknowledgments This work is supported by the European Union through the MSCA SE project QCOMICAL (Grant Agreement ID: 101182520), by the Plan France 2030 through the PEPR integrated project EPiQ (ANR-22-PETQ-0007), and by the Uruguayan CSIC grant 22520220100073UD.

References

[BvN36] Garrett Birkhoff and John von Neumann. The logic of quantum mechanics. *Annals of Mathematics*, 37(4):823–843, 1936. doi:10.2307/1968621.

[Cro93] Roy Crole. *Categories for Types*. Cambridge University Press, 1993. doi:10.1017/CBO9781139172707.

[DC25] Alejandro Díaz-Caro. Towards a computational quantum logic: An overview of an ongoing research program. Invited talk at the Quantum Computing session at CiE 2025: Computability in Europe. To appear at LNCS. Preprint at arXiv:2504.07609, 2025.

[DCD23] Alejandro Díaz-Caro and Gilles Dowek. A new connective in natural deduction, and its application to quantum computing. *Theoretical Computer Science*, 957:113840, 2023. doi:10.1016/j.tcs.2023.113840.

[DCD24] Alejandro Díaz-Caro and Gilles Dowek. A linear linear lambda-calculus. *Mathematical Structures in Computer Science*, 34:1103–1137, 2024. doi:10.1017/S0960129524000197.

[DCD25] Alejandro Díaz-Caro and Gilles Dowek. A new introduction rule for disjunction. arXiv:2502.19172, 2025.

[DCDIM24] Alejandro Díaz-Caro, Gilles Dowek, Malena Ivnisky, and Octavio Malherbe. A linear proof language for second-order intuitionistic linear logic. In George Metcalfe, Thomas Studer, and Ruy de Queiroz, editors, *Logic, Language, Information and Computation (WoLLIC 2024)*, volume 14672 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2024. doi:10.1007/978-3-031-62687-6_2.

[DCDR19] Alejandro Díaz-Caro, Gilles Dowek, and Juan Pablo Rinaldi. Two linearities for quantum computing in the lambda calculus. *BioSystems*, 186:104012, 2019. Postproceedings of TPNC 2017. doi:10.1016/j.biosystems.2019.104012.

[DCGMV19] Alejandro Díaz-Caro, Mauricio Guillermo, Alexandre Miquel, and Benoît Valiron. Realizability in the unitary sphere. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019)*, pages 1–13, 2019. doi:10.1109/LICS.2019.8785834.

[DCM20] Alejandro Díaz-Caro and Octavio Malherbe. A categorical construction for the computational definition of vector spaces. *Applied Categorical Structures*, 28(5):807–844, 2020. doi:10.1007/s10485-020-09598-7.

[DCM22] Alejandro Díaz-Caro and Octavio Malherbe. Quantum control in the unitary sphere: Lambda-S_1 and its categorical model. *Logical Methods in Computer Science*, 18(3:32), 2022. doi:10.46298/lmcs-18(3:32)2022.

[DCM23] Alejandro Díaz-Caro and Octavio Malherbe. A concrete model for a linear algebraic lambda calculus. *Mathematical Structures in Computer Science*, 34(1):1–44, 2023. doi:10.1017/s0960129523000361.

[DCM24] Alejandro Díaz-Caro and Octavio Malherbe. The sup connective in IMALL: A categorical semantics. arXiv:2405.02142, 2024.

[Mon25] Nicolás Monzón. Extension of Lambda-S for different measurement bases. Master's thesis, Universidad Argentina de la Empresa, March 2025.

[SU06] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006. doi:10.1016/S0049-237X(06)80499-4.

Internal Proofs of Strong Normalization

Cody Roux

AWS

Strong normalization (henceforth SN) is a useful property for a type theory to have. Among other things, it usually provides

1. consistency of the system under scrutiny
2. the existence of canonical forms for every term
3. decidability of type checking

of course, this is not the only way to provide such guarantees. For consistency, any model will suffice, and decidability of type checking, while quite nice in theory, is generally accompanied by potential runtimes that strain the meaning of the word. Canonicity can also be achieved nowadays via more subtle methods, like normalization-by-evaluation (NbE), which may be available in cases where SN may not hold, e.g. in the presence of certain η-laws.

However, SN does have the advantage of being straightforward to state, and very effective, leaving to the implementer the choice of normalization strategy, which is mostly delegated to the meta-theory in the case of NbE. We also regard the proof of normalization to have a certain combinatorial charm, with certain questions around it, notably the Barendregt-Geuvers-Klop conjecture (see, e.g. Barthe & al [2]), being still open.

Most proofs of normalization involve a certain kind of model construction, the so-called *logical relations model*. This is unsurprising based on point 1 above, for Gödelian reasons, as the proof SN $\Rightarrow$ consistency can often be carried out in a very weak system (and therefore the "tough bit" must be in the proof of normalization). This construction feels very similar to the classic realizability interpretation $t \Vdash P$ of Kleene which relates certain kinds of program encodings t to propositions P, and indeed may be seen as a generalization of it.

It is a classical fact that, in Heyting arithmetic HA, any theorem P has a realizer t, and that HA "knows" that it is a realizer:

$$HA \vdash P \quad \Rightarrow \quad HA \vdash \ulcorner t \Vdash P \urcorner$$

This is nice, but the richness of HA allows for a lot of mess in this proof, e.g. there may be some instances of induction required to carry out the second proof that are not obviously related to the first, to deal with encoding issues.

We show that this is not necessarily the case, by working in a setting potentially much poorer than HA, at least in terms of expressiveness, and adding only the requisite tools to express and prove the strong normalization theorem (for a given well-typed term) without any additional meta-theoretic tools.

It's worth noting that such a proof translation will not really guarantee normalization, of course: we need an additional fact, namely *existence of normal forms*. This may seem redundant, but it is worth noting that existence of normal forms is actually weaker than strong normalization! In essence our strategy is to bootstrap this stronger property from the weaker one.

Our proof is quite similar to that of Berger & al [3] with the main difference being that we aim to identify exactly which type theoretic mechanisms are required in the target logic,

with the aim of identifying a natural map from theories to theories which can encode their own normalization proof.

To translate the proof, move from a source type theory $\mathcal{T}$ to a richer theory $\hat{\mathcal{T}}$ which can express facts about syntax.

We will need a new sort which we call Syn. This sort will contain a number of types (Term, Var, ...), which themselves contain symbols for terms, variables, substitutions etc. as well as constructors for the normalization predicates, SN and neutral terms NE essential to the proof. We do *not* need eliminators for any of these types, since induction shall (and must!) remain at the meta-level.

We further extend the target type systems with a dependent product $\Pi x : T.U$ in the case that T : Syn is a syntax form and U is a well typed type in the original system, e.g. has type Type or Prop. The resulting product is in the sort of T, allowing us to chain such products. We note here that the fact that SN and NE eliminate into that sort as well requires us to have at least a non-dependent function type of that sort, that is, if $T,\ U$: Type then $T \to U$: Type.

Finally we extend the normal conversion rule of the theory, by adding rewrite rules over the inhabitants of Syn to encode *substitution*, which can be taken, e.g. to be the σ-rules of Abadi & al [1]. However, we do *not* add a β-rule for syntactic terms. Intuitively, these rules allow us to reason modulo substitutions, as they are confluent and terminating in the absence of β.

Now we can translate term derivations in the source theory into derivations of computability in the target theory. The idea is to translate each type T : Type (or any other sort) into a predicate $[\![T]\!]$: Term $\to$ Type, and prove, by (meta-)induction on the type derivation,

$$\Gamma \vdash_{\mathcal{T}} t : T \quad \Rightarrow \quad [\![\Gamma]\!] \vdash_{\hat{\mathcal{T}}} (\!|t|\!) : [\![T]\!]\ \ulcorner t^o \urcorner$$

where $\ulcorner t^o \urcorner$ is the *open* syntactic form of t, containing variables of type Term appearing in $[\![\Gamma]\!]$ which correspond to the original t variables. Predictably, the translation of non-dependent arrow types will be the "realizability arrow"

$$[\![A \to B]\!] := \lambda t : \text{Term}.\Pi u : \text{Term}.[\![A]\!]\ u \to [\![B]\!]\ (t\ u)$$

but type variables X will need their own variable predicates $\mathcal{X}$, along with additional constructors

$$\text{ne} : \Pi t : \text{Term}.\text{NE}\ t \to \mathcal{X}\ t$$

$$\text{sn} : \Pi t : \text{Term}.\mathcal{X}\ t \to \text{SN}\ t$$

$$\text{wh} : \Pi t\ u\ t' : \text{Term}.\text{SN}\ u \to \text{WHExp}\ t\ t'\ u \to \mathcal{X}\ t' \to \mathcal{X}\ t$$

The predicate WHExp expresses that t is the weak head expansion of t' by u, that is, $t = (\lambda x.t_1)\ u\ t_2 \dots t_n$ and $t' = t_1[u/x]\ t_2 \dots t_n$.

These conditions are exactly the *Girard conditions* (at least, the so-called "saturated sets" form), and sadly, we must add with them some common sense reductions. Thankfully these conversions do not interact with β-reductions, and so do not threaten weak normalization.

Finally, from there we build a proof $\mathcal{G} \vdash \text{sn}_t : \text{SN}\ \ulcorner t \urcorner$ for a well chosen $\mathcal{G}$ (closely related to Γ), a surprisingly non-trivial task in the absence of induction over syntax! This time, $\ulcorner t \urcorner$ is the actual syntax of t, variables and all. From there we may conclude, if the original type theory admits normal forms, that t is indeed strongly normalizing. This relies on a simple induction on normal forms of terms of type SN t, along with the following meta-theorem:

Theorem 1. *If $\mathcal{T}$ admits normal forms, so does $\hat{\mathcal{T}}$.*

The feasibility of our translation, and of course, the correctness of our final conclusion (that t is strongly normalizing) will depend on exactly which constructors we give for SN, and whether they are true in the meta-theory. Here, again, we proceed similarly to Berger & al [3] and add the (folklore) inductive characterizations of strong normalization using weak-head expansions, and an additional fact: SN t holds if SN $(t\ x)$ holds for any variable x.

We give, as illustration, a representation of the normal form of the proof of SN for $(\lambda x.x)\ y$, which, while not eminently surprising, may illustrate the strategy.

$$\mathcal{A} : \mathrm{Term} \to *,$$

$$\mathrm{ne}_{\mathcal{A}} : \Pi t : \mathrm{Term}.\mathrm{NE}\ t\ \to \mathcal{A}\ t,\ \mathrm{sn}_{\mathcal{A}} : \Pi t : \mathrm{Term}.\mathcal{A}\ t \to \mathrm{SN}\ t,$$

$$\mathrm{wh}_{\mathcal{A}} : \Pi t\ t'\ u.\mathrm{WHExp}\ t\ t'\ u \to \mathrm{SN}\ u \to \mathcal{A}\ t' \to \mathcal{A}\ t,$$

$$\vdash$$

$$\mathrm{wh}_{\mathrm{SN}}\ \ulcorner(\lambda x.x)\ y\urcorner\ \ulcorner y\urcorner\ \ulcorner y\urcorner\ (\mathrm{ne}_{\mathrm{SN}}\ (\mathrm{ne}_{\mathrm{Var}}\ulcorner y\urcorner))\ (\mathrm{ne}_{\mathrm{SN}}\ (\mathrm{ne}_{\mathrm{Var}}\ulcorner y\urcorner)) : \mathrm{SN}\ \ulcorner(\lambda x.x)\ y\urcorner$$

here $\mathrm{wh}_{\mathrm{SN}}$, $\mathrm{ne}_{\mathrm{SN}}$, $\mathrm{ne}_{\mathrm{Var}}$ etc. are part of the constructors for the type families SN and NE, and as a result the normalization of the top-level term can simply be read off of the proof, and the occurrences of $\mathrm{wh}_{\mathcal{A}}$ were reduced away as the semantic variable y was replaced by its syntactic counterpart $\ulcorner y\urcorner$.

As a conclusion, we note that the idea of conducting a logical relations argument within its own type theory is not new; our work is inspired from Bernardy & Lasson [4], where they introduce, for a given type theory, an enriched theory capable of expressing logical relations between the terms of the original theory.

There are many similar such categorical investigations (see e.g. Bocquet & al [5, 6]), though the author is at the moment incapable of determining the exact relationship with this more syntactic argument. The main novelty here is to carry out the combinatorial argument for normalization internally, including the last step of "extracting" the normal form.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. Weak normalization implies strong normalization in a class of non-dependent pure type systems. *Theoretical Computer Science*, 269(1):317–361, 2001.

[3] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica: An International Journal for Symbolic Logic*, 82(1):25–49, 2006.

[4] Jean-Philippe Bernardy and Marc Lasson. Realizability and parametricity in pure type systems. In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures*, pages 108–122, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. For the metatheory of type theory, internal sconing is enough, 2023.

[6] Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory.* PhD thesis, Carnegie Mellon University, 2022.

Compositional memory management in the λ-calculus

Sky Wilshaw[1] and Graham Hutton[2]

[1] School of Computer Science, University of Nottingham
sky.wilshaw@nottingham.ac.uk
[2] School of Computer Science, University of Nottingham
graham.hutton@nottingham.ac.uk

1 Motivation and overview

Type systems can be used in programming languages to guarantee memory safety. This idea has been developed in many different directions, for example with *borrowing* from the Rust language [6, 3], the *reachability types* of Bao et al. [1], and the *fractional uniqueness* types of Marshall and Orchard [5]. In order to create new type systems of this form in a principled way, we would like a simple untyped system that supports a basic form of memory management, so that type systems can be built on top of it. However, existing untyped languages of this kind typically suffer from two major problems. First, they often introduce many new primitives for concepts such as memory cells, allocation, freeing, pointers, and stores, complicating the semantics. Second, they are usually non-compositional, as we need to thread the state of the memory store through all derivations; this is a particular weakness when our goal is to design type systems.

In this work, we identify a fragment of manual memory management, which we call *explicit naming*. Under this paradigm, names are first-class citizens in a language, with explicit primitives for creating, using, and freeing names. These primitives correspond to simple operations on memory, interpreting names as pointers to the values they are bound to. We will introduce a lambda calculus with explicit naming, with a stateful semantics that keeps track of the value assigned to each name. Then, to address the lack of compositionality, we will develop a compositional semantics for the same language, and show that it is equivalent to its non-compositional counterpart. This language, with its two equivalent semantics, provides a framework that can be used to develop type systems for explicit naming.

2 Modifying the lambda calculus

We will start with a simple call-by-value lambda calculus, with standard reduction rules and notation inspired by Launchbury's semantics for lazy evaluation [4]. We use the standard syntax for lambda terms, writing expressions with the letter e and values with the letter v. Judgments are of the form $H_1 : e \Downarrow H_2 : v$, where H_1 and H_2 are *heaps*, finite partial functions from names to values. This can be read as 'expression e can be evaluated with heap H_1 to produce a value v and a resulting heap H_2'.

Our initial rules are as follows. Problems relating to name collisions and α-equivalence will not be discussed here in order to focus on the key ideas of this contribution. Colours are not syntactically important.

$$\frac{}{(H, x \mapsto v) : x \Downarrow (H, x \mapsto v) : v}\ \textsc{Var}' \qquad \frac{}{H : \lambda x.\, e \Downarrow H : \lambda x.\, e}\ \textsc{Lam}$$

$$\frac{H_1 : e_1 \Downarrow H_2 : \lambda x.\, e \qquad H_2 : e_2 \Downarrow H_3 : v \qquad (H_3, x \mapsto v) : e \Downarrow H_4 : v'}{H_1 : e_1\ e_2 \Downarrow H_4 : v'}\ \textsc{App}$$

Here, the variable rule Var′ tells us that a name x evaluates to the value v bound to it in the heap. This means that every *mention* of a name is a *use* of it, or alternatively, that we cannot have a pointer without dereferencing it. This prevents us from using names as first-class citizens. Therefore, to make a lambda calculus with explicit naming, we need to divide the variable rule into the following two rules:

$$\frac{}{H : x \Downarrow H : x}\ \textsc{Var} \qquad \frac{H_1 : e \Downarrow (H_2, x \mapsto v) : x}{H_1 : !e \Downarrow (H_2, x \mapsto v) : v}\ \textsc{Read}$$

We have introduced a new operator, written '!', for looking up the value bound to a name. With these new rules, names are already values, and do not reduce; a name must be explicitly dereferenced using '!' in order to access its value.

To model explicit naming, it remains to provide a method to free names. This allows us to model one of the key challenges with memory management, namely, that dereferences might fail after a deallocation. To do this, we add in a second new primitive, which evaluates an expression and then frees a name (which itself may be the result of evaluating some other expression).

$$\frac{H_1 : e_1 \Downarrow H_2 : v \qquad H_2 : e_2 \Downarrow (H_3, x \mapsto v') : x}{H_1 : e_1; \text{ free } e_2 \Downarrow H_3 : v}\ \textsc{Free}$$

3 Recovering compositionality

The main disadvantage of the presented semantics is its non-compositionality, as the heap state must be threaded through each derivation. This prevents us from building type systems on this language. To combat this, we will describe a new semantics which calculates the *denotation* of an expression separately from computing the *effect* on the heap of evaluating this expression. Crucially, we do not need to compute intermediate heap states in order to calculate denotations or effects of compound expressions; it is in this sense that our effect-based semantics is compositional.

In order to do this, we need to reduce our dependence on the heap state, by moving some of the information from the heap into the values. In particular, abstractions now evaluate to closures, and variables now keep track of their assigned values. These new values are called *denotational values* and written with the letter w. Rather than in a heap, denotational values are stored in a *context* Γ, a finite partial function from names to denotational values.

To be precise, a denotational value is either a closure of the form $(\lambda^{\Gamma} x.\, e)$ or a variable binding $(x \mapsto w)$. Such a denotational value can be thought of as a value in the usual sense, together with all of the data that it could 'see' in the heap at the time of its creation. We define denotational values coinductively to allow for loops in variable bindings and stored contexts.

Our big-step reduction relation will now have the form $\Gamma \vdash e \Downarrow f : w$, where f is the *effect*, a partial function from heaps to heaps, and w is a denotational value. We may view f as the effect on the heap of evaluating $\Gamma \vdash e \Downarrow w$. Such partial functions evidently form a monoid under composition.

$$\frac{}{(\Gamma, x \mapsto w) \vdash x \Downarrow \mathrm{id} : (x \mapsto w)}\ \text{F-Var} \qquad \frac{}{\Gamma \vdash \lambda x.\, e \Downarrow \mathrm{id} : \lambda^{\Gamma} x.\, e}\ \text{F-Lam}$$

$$\frac{\Gamma \vdash e_1 \Downarrow f_1 : \lambda^{\Gamma'} x.\, e \qquad \Gamma \vdash e_2 \Downarrow f_2 : w \qquad \Gamma', x \mapsto w \vdash e \Downarrow f_3 : w'}{\Gamma \vdash e_1\ e_2 \Downarrow f_3 \circ (x := w) \circ f_2 \circ f_1 : w'}\ \text{F-App}$$

$$\frac{\Gamma \vdash e \Downarrow f : (x \mapsto w)}{\Gamma \vdash !e \Downarrow \mathrm{read}\, x \circ f : w}\ \text{F-Read} \qquad \frac{\Gamma \vdash e_1 \Downarrow f_1 : w \qquad \Gamma \vdash e_2 \Downarrow f_2 : (x \mapsto w')}{\Gamma \vdash e_1; \mathrm{free}\ e_2 \Downarrow \mathrm{free}\, x \circ f_2 \circ f_1 : w}\ \text{F-Free}$$

Here, $(x := w)$, $\mathrm{read}\, x$, $\mathrm{free}\, x$ denote partial functions that perform the given operation. For instance, $(x := w)$ adds the binding $x \mapsto w$ to any heap where x is not in its domain, and is undefined elsewhere. Similarly, $\mathrm{read}\, x$ is the identity on heaps that contain x in their domain, and undefined elsewhere. In these rules, effects of subexpressions are only used to compute the composite effect of the expression, and are not applied to contexts or used serially between subexpressions. We do not use the full flexibility of partial functions between heaps, and various simpler presentations of our effects are possible, for example using *effect quantales* [2].

4 Equivalence

We can show that the two semantics are equivalent. To formalise this equivalence, we need to set up some conversions between heaps and contexts. First, we can 'forget' the extra data of a denotational value w to recover a value $v = \overline{w}$. This operation is defined by $\overline{(\lambda^{\Gamma} x.\, e)} = \lambda x.\, e$ and $\overline{(x \mapsto w)} = x$. In the other direction, we can translate a heap H into a context $\Gamma = \mathrm{tr}(H)$ defined by the following coinductive rules.

$$\frac{H(x) = y \qquad \mathrm{tr}(H)(y) = w}{\mathrm{tr}(H)(x) = (y \mapsto w)} \qquad \frac{H(x) = \lambda y.\, e}{\mathrm{tr}(H)(x) = \lambda^{\mathrm{tr}(H)} y.\, e}$$

We can then prove the following result.

Theorem. *Let e be an expression and H be a heap. We define inductively that a variable x is* reachable *from e in H if it is free in e or is reachable from $H(y)$ where y is free in e. If all variables reachable from e in H occur in the domain of H, then*

$$H : e \Downarrow H' : v \iff (\exists w\, f,\ \overline{w} = v \ \wedge\ f(H) = H' \ \wedge\ \mathrm{tr}(H) \vdash e \Downarrow f : w)$$

where the derivations on the left and right sides of this equivalence pull from the same stream of fresh names.

The hypothesis of this theorem is a minor generalisation of the usual notion of an 'expression-in-context', ensuring that $\mathrm{tr}(H)$ contains all of the free variables of e.

This equivalence theorem allows for more compositional reasoning about memory-sensitive computation. As an example, we can show that if two expressions e_1 and e_2 have disjoint sets of reachable names in H, then they can be evaluated in either order without affecting the overall computation. This is because the effects f_1 and f_2 of evaluating these expressions in $\mathrm{tr}(H)$ operate on disjoint sets of names and therefore commute past one another: $f_1 \circ f_2 = f_2 \circ f_1$.

We hope that this compositional framework is suitable for developing interesting type theories that are capable of encapsulating some of the challenges of memory management.

References

[1] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–32, 2021.

[2] Colin S. Gordon. Polymorphic iterable sequential effect systems. *ACM Trans. Program. Lang. Syst.*, 43(1), April 2021.

[3] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[4] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 144–154, New York, NY, USA, 1993. Association for Computing Machinery.

[5] Danielle Marshall and Dominic Orchard. Functional ownership through fractional uniqueness. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024.

[6] Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, October 2014.

Choice principles and a cotopological modality in HoTT

Owen Milner

Department of Philosophy, Carnegie Mellon University

Anel and Barton [1] define a family of generalized forms of the axiom of countable choice, and discuss their relationship to Postnikov completeness and the hypercompletion operation in ∞-toposes. This talk will discuss the translation of their work into HoTT and a formalization in Cubical Agda. In particular, we will present a construction of the hypercompletion operation in HoTT under the assumption that any one of those forms of the axiom of countable choice holds.

Working in intensional Martin-Löf type theory with a univalent universe, we say that a type is (-2)-truncated if it is equal to the unit type. A type is called $(n+1)$-truncated ($-2 \leq n < \infty$) when its identity types are n-truncated. For any $n < \infty$, there is an operation $\lambda X.\|X\|_n$ sending a type to its *n-truncation* – which is a *universal* n-truncated type with a map $|\cdot|_n : X \to \|X\|_n$.

We say that a modality is any operation $\bigcirc : \mathcal{U} \to \mathcal{U}$ together with a *unit* $\eta : (X : \mathcal{U}) \to X \to \bigcirc X$, which satisfy certain conditions (see [4] for the definitive account). If η_X is an equivalence, we say that X is a modal type for the modality $(\bigcirc, \eta)$. The n-truncation operation, together with the universal map, is a modality, its modal types are precisely the n-truncated types.

A type X is called n-connected if $\|X\|_n$ is equal to the unit type. A type Y is n-truncated if and only if, for all n-connected types X, the canonical map $Y \to (X \to Y)$ is an equivalence. We say that a type is ∞-connected if $\|X\|_n$ is equal to the unit type *for all* n. By analogy with the finite case, we say that a type Y is ∞-truncated if, for all ∞-connected types X, the canonical map $Y \to (X \to Y)$ is an equivalence.

An important construction in ∞-topos theory is the *hypercompletion* operation, which sends each object in an ∞-topos to an ∞-truncated reflection of that object [2]. Since HoTT has semantics in ∞-toposes [5], it is desirable to construct such an operation in the type theory. In particular, this ought to be a modality whose modal types are precisely the ∞-truncated types. However, the authors of [4] remark that it is unknown how to construct such a modality in HoTT without additional assumptions. As far as I am aware, little subsequent progress has been made on the problem.

That said, we clearly can construct this modality under particular assumptions. In particular – if we assume that each type is already ∞-truncated, then the hypercompletion agrees with the trivial modality. However, as we're going to see, this is not the only condition which allows us to construct this modality.

Postnikov towers

A formal Postnikov tower is a family of types $A : \mathbb{N} \to \mathcal{U}$, with a family of maps $a : (n : \mathbb{N}) \to A_{n+1} \to A_n$, such that for each $n : \mathbb{N}$, A_n is n-truncated and a_n has n-connected fibers. For every type X there is a canonical Postnikov tower with $A_n := \|X\|_n$ and $a_n := |\cdot|_n$. For any

tower (Postnikov or otherwise) we can define its limit:

$$\lim(A, a) := (x : (n : \mathbb{N}) \to A_n) \times ((n : \mathbb{N}) \to a_n(x_{n+1}) = x_n)$$

There is a canonical morphism from a type X into the limit of its canonical Postnikov tower. Postnikov convergence is the statement that these canonical morphisms are always equivalences. Meanwhile, if (A, a) is a Postnikov tower, there is a canonical family of maps $\epsilon_n : \| \lim(A, a)\|_n \to A_n$, and Postnikov effectiveness is the statement that for every Postnikov tower, each ϵ_n is an equivalence. The conjunction of Postnikov convergence and Postnikov effectiveness is called Postnikov completeness.

Postnikov convergence implies that every type is ∞-truncated – so, again, the hypercompletion modality agrees with the trivial modality if Postnikov convergence holds. Postnikov effectiveness also allows us to construct a hypercompletion modality and in this case (so long as Postnikov convergence does not also hold) it is non-trivial – the operation sends a type to the limit of its canonical Postnikov tower:

$$\lambda X. \lim(\|X\|_n, |\cdot|_n)$$

So far so good, but actually we can get away with a simpler assumption, one which does not refer to Postnikov towers. Anel and Barton [1] define countable choice of dimension $\leq d$ for a fixed ∞-topos to be the principle stating that if $X_1, X_2, \ldots$ form a family of $d + k$-connected objects, then their product $\Pi_{\mathbb{N}} X_n$ is k-connected. They show that each one of these principles implies (an external version of) Postnikov effectiveness for the topos. They observe that their definitions and most of their arguments can be reformulated in HoTT. For a formalization in Cubical Agda see the repository: [3]. All of this allows us to conclude in HoTT that for any d, countable choice of dimension $\leq d$ allows us to construct a hypercompletion modality.

Acknowledgements

I am grateful to Mathieu Anel and Reid Barton for discussing their work with me, and encouraging me to work on the formalization. I am also grateful to them and to Steve Awodey for reading and commenting on a draft of this abstract. I have given talks on this work previously for the CMU HoTT research seminar, and for the University of Nottingham functional programming group lunch, I am grateful to the organizers of both. And to several anonymous reviewers for their kind and helpful recommendations.

This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-21-1-0009, PI Steve Awodey.

References

[1] Mathieu Anel and Reid Barton. "Choice axioms and Postnikov completeness". 2024. URL: https://arxiv.org/abs/2403.19772.
[2] Jacob Lurie. *Higher Topos Theory*. Princeton University Press, 2009.
[3] Owen Milner. Github Repository. URL: https://github.com/owen-milner/choicepostnikov.
[4] Egbert Rijke, Michael Shulman, and Bas Spitters. "Modalities in homotopy type theory". In: *Logical Methods in Computer Science* 16.1 (2020).

[5] Michael Shulman. "All $(\infty, 1)$-toposes have strict univalent universes". 2019. URL: https://arxiv.org/abs/1904.07004.

Towards Modular Composition of Inductive Types Using Lean Meta-programming

Ramy Shahin

Qualgebra
ramy@qualgebra.com

Introduction Inductive types are ubiquitous building blocks in many programming and theorem proving languages. An inductive type is a closed set of constructors from which values of the type can be created. That set of constructors cannot be extended though once a type is defined. This limits extensibility, reuse, and modular separation of concerns when defining types and functions operating over their values. The *expression problem* [13] is one manifestation of this limitation, where extending an expression language with new syntactic constructors while reusing existing ones without having to modify or re-compile them is a challenge in almost all programming languages.

This extended abstract briefly presents a set of syntactic extensions to the Lean programming language that allow the modular compositions of inductive types[1]. Lean 4 [7] includes meta-programming constructs that allow developers to extend the syntax of the language, and provide user-defined elaborators of the extended syntactic constructs. We utilize those meta-programming facilities to allow the definition of inductive types that are built from *clones* of the constructors of constituent types. In addition, we automatically generate coercion operators that allow passing values of constituent types to functions expecting values of the extended type, and vice versa when applicable.

```
namespace Boolean

inductive T where
| Bool

inductive Term where
| True
| False
| If (c t₁ t₂: Term)

inductive TRel: Term → T → Prop
| TT: TRel .True .Bool
| FF: TRel .False .Bool
| If: TRel c .Bool → TRel t₁ τ → TRel t₂ τ
    → TRel (.If c t₁ t₂) τ

end Boolean
```

(a) Boolean type definition.

```
namespace Nat

inductive T where
| N

inductive Term where
| Zero
| Succ (t: Term)
| Pred (t: Term)

inductive TRel: Term → T → Prop where
| Z: TRel .Zero .N
| S: TRel t .N → TRel (.Succ t) .N
| P: TRel t .N → TRel (.Pred t) .N

end Nat
```

(b) Nat type definition.

Figure 1: Separate definitions of `Boolean` and `Nat` types, syntactic terms, and type relation.

[1] Prototype implementation can be found at https://github.com/qualgebra/LeanToolkit/tree/TYPES2025

```
inductive T := Boolean.T + Nat.T
```

⇓

```
inductive T : Type
| Bool : T
| N : T
```

(a) Composing `Boolean.T` and `Nat.T`.

```
inductive Term := Boolean.Term + Nat.Term
| isZero (t: Term)
```

⇓

```
inductive Term : Type
| True : Term
| False : Term
| If : Term → Term → Term → Term
| Zero : Term
| Succ : Term → Term
| Pred : Term → Term
| isZero : Term → Term
```

(b) Composing `Boolean.Term` and `Nat.Term`.

```
inductive TRel: Term → T → Prop := Boolean.TRel + Nat.TRel
| iz: TRel t T.N → TRel (.isZero t) T.Bool
```

⇓

```
inductive TRel : Term → T → Prop
| TT: TRel Term.True T.Bool
| FF: TRel Term.False T.Bool
| If: ∀{c t₁: Term} {τ:T} {t₂: Term}, TRel c T.Bool → TRel t₁ τ → TRel t₂ τ → TRel (c.If t₁ t₂) τ
| Z: TRel Term.Zero T.N
| S: ∀ {t : Term}, TRel t T.N → TRel t.Succ T.N
| P: ∀ {t : Term}, TRel t T.N → TRel t.Pred T.N
| iz: ∀ {t : Term}, TRel t T.N → TRel t.isZero T.Bool
```

(c) Composing `Boolean.TRel` and `Nat.TRel`.

Figure 2: Composing `Boolean` and `Nat` using the '+' operator on inductive types.

Example This example is inspired by the Typed Lambda Calculus (TLC) presentation from [8]. We assume we are defining TLC with two native types: `Boolean` (Fig. 1a), and `Nat` (Fig. 1b). Each of the two separate definitions includes inductive types for the set of relevant types (`T`), the set of valid syntactic terms (`Term`), and a type relation (`TRel`) between terms and types.

Now we would like to compose the two sets of definitions into a language with both `Boolean` and `Nat` native types. We use Lean meta-programming to extend the Lean syntax with a new construct for *summing up* multiple inductive types. Fig. 2 shows three examples: composing `Boolean.T` and `Nat.T` (Fig. 2a), composing `Boolean.Term` and `Nat.Term`, while adding an extra constructor `isZero` (Fig. 2b), and finally composing `Boolean.TRel` and `Nat.TRel`, adding the extra constructor `iz` (Fig. 2c). The '+' operator (implemented and elaborated using Lean meta-programming) is used in all three examples to compose multiple inductive types, and optionally adding extra constructors like in the cases of `Term` and `TRel`. Each of the examples shows the Lean definition automatically generated as a result.

In addition, instances of the `Coe` typeclass are also generated to allow safe automatic coercion from values of the constituent types to the newly defined composite type. For example, the following code snippet typechecks because the automatically generated coercion operator converts `Boolean.T.Bool` into T.Bool:

```
def x := Boolean.T.Bool
def y: T := x
```

Coercion in the opposite direction (e.g., from `T.Bool` to `Boolean.T.Bool`) is possible only for values known at compile time. The Lean standard library includes the dependent coercion typeclass `CoeDep`. We generate instances of this typeclass for each of the constructors of the summed up type, coercing them back to their respective constituent types. As a result, the following Lean definition typechecks:

```
def z: Boolean.T := T.Bool
```

Related Work Previous work tried to reuse proofs on modular definitions in Coq [2, 9], by extending an inductive type by individual extra constructors, and functions with individual pattern matching cases. Modular composition of definitions and theorems into feature-based product lines was presented in [3]. Inspired by the data types a-la-carte work for Haskell [11], similar approaches to solving the expression problem in theorem provers include Meta-theory a la carte [4], Coq-a-la-carte [5], and extensible metatheory mechanization [6]. Other attempts at solving the expression problem include four different solutions relying on generic data types in Java-like languages are presented in [12], and a symmetric view of algebraic data types and codata types [1].

Our approach of composing constructors is similar to that of Boite [2] with three main differences that we know of.

1. Boite's approach incrementally adds constructors to an existing type, while we focus on composing multiple types, and also support adding extra constructors if needed. We also rely on coercion operators for interoperation between composed and constituent types.
2. We heavily leverage Lean metaprogramming to simplify the implementation, while Boite's work predates MetaCoq [10].
3. This is more of a limitation on our side at this point, we do not support composing proof objects. This is one of our future work directions.

Acknowledgments The author would like to thank anonymous reviewers for their feedback and insightful suggestions.

References

[1] David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. Decomposition diversity with symmetric data and codata. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[2] Olivier Boite. Proof Reuse with Extended Inductive Types. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics*, pages 50–65, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[3] Benjamin Delaware, William Cook, and Don Batory. Product Lines of Theorems. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 595–608, New York, NY, USA, 2011. ACM.

[4] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. *SIGPLAN Not.*, 48(1):207–218, January 2013.

[5] Yannick Forster and Kathrin Stark. Coq à la carte: a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 186–200, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Ende Jin, Nada Amin, and Yizhou Zhang. Extensible metatheory mechanization via family polymorphism. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[7] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, page 625–635, Cham, 2021. Springer International Publishing.

[8] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[9] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for Proof Reuse in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[10] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

[11] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[12] Mads Torgersen. The expression problem revisited. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, pages 123–146, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[13] Philip Wadler. The Expression Problem. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt. *Note to Java Genericity mailing list.*

Type-safe Bidirectional Channels in Idris 2

Guillaume Allais[1]

University of Strathclyde, Glasgow, Scotland, United Kingdom
guillaume.allais@strath.ac.uk

Introduction

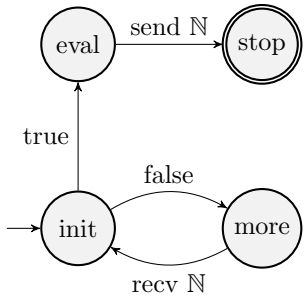
Session types are a channel typing discipline ensuring that the communication patterns of concurrent programs abide by a shared protocol. A meta-theoretical analysis of the typing discipline can then ensure that the communicating processes will have good properties e.g. deadlock-freedom. We show how to use a linear dependently typed programming language to define a direct and type-safe embedding of expressive binary session types.

Expressive Session Types

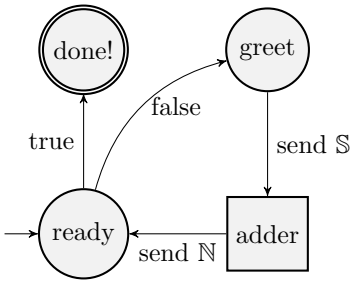
Our session types are closed under nested smallest ($\mu x.\ \cdot$) and largest ($\nu x.\ \cdot$) fixpoints, typed sends ($!A.\ \cdot$) and receives ($?A.\ \cdot$), the empty protocol (`end`) and binary branch offerings ($\cdot \,\&\, \cdot$) and selections ($\cdot \oplus \cdot$).

$$\begin{aligned} p,\ q,\ \ldots ::= {} & x \mid \mu x.\ p \mid \nu x.\ p \\ & \mid\ !A.\ p \mid ?A.\ p \mid \texttt{end} \\ & \mid p \,\&\, q \mid p \oplus q \end{aligned}$$

The session $\nu i.\ ((!\mathbb{N}.\ \texttt{end}) \,\&\, (?\mathbb{N}.\ i))$ types a server for the 'adder' protocol represented as a finite state machine on the right hand side. The server accepts an arbitrarily long sequence of natural numbers (sent by a client repeatedly selecting the *more* $= (?\mathbb{N}.\ x)$ branch) before sending back their sum as a single number (when the *eval* $= (!\mathbb{N}.\ \texttt{end})$ branch is selected) and terminating. The encoding of the server's type uses a largest fixpoint because the client is the one responsible for ensuring the process terminates by eventually selecting the *eval* branch.



We can use nested fixpoint to describe more complex protocols. For instance, the session described by $\nu r.\ (\texttt{end} \,\&\, (!\mathbb{S}.\ \nu i.\ ((!\mathbb{N}.\ r) \,\&\, (?\mathbb{N}.\ i))))$ informally corresponds to the finite state machine shown on the right hand side. It types a server repeatedly offering to handle the adder protocol described above after greeting the user with a string. We replaced adder's `end` with r to let the protocol loop back to the 'ready' state once the adder is done (in the state machine, that corresponds to merging 'stop' and 'ready' hence the bottom 'send ℕ'-labeled arrow from the 'adder' sub-machine back to 'ready').



Type-safe Implementation

Pairs of Unityped Channels We represent a bidirectional channel parametrised by a session type as a pair of unityped unidirectional channels. The key idea behind this safe implementation is to collect all the received (respectively sent) types used in a protocol and to manufacture a big sum type in which we can inject all of the received (respectively sent) values. Because

the received types of a protocol are equal to the sent types of its dual (and vice-versa), we can prove that the directed channels of two communicating parties are indeed type-aligned. The definition of this sum type is a type safe realisation of Kiselyov and Sabry's open union type [5] using the encoding of scope-safe De Bruijn indices [4] introduced by Brady in Idris 2's core [3]. We include its definition below: given a list `ts` of values of type `a` and a decoding function `elt` turning such values into types, (`UnionT elt ts`) defines a union type. Its one constructor named `Element` takes a natural number `k` functioning as a tag, a proof of type (`AtIndex t ts k`) that the value `t` is present at index `k` in `ts` and injects a value of type (`elt t`) into the union. By picking `elt` to be the identity function, we recover the usual notion of tagged union of types.

```
data UnionT : (elt : a -> Type) -> (ts : List a) -> Type where
  Element : (k : Nat) -> (0 _ : AtIndex t ts k) -> elt t -> UnionT elt ts
```

Note that the proof of type (`AtIndex t ts k`) is marked as erased through the use of the `0` quantity which means this datatype compiles down to a pair of a GMP-style `Integer` (the optimised runtime representation of the tag of type `Nat`) and a value of the appropriate type. This gives us an encoding that has constant time injections and partial projections (via an equality check on the `Nat` tag) in and out of the big sum type.

Dealing with Changing Sessions The session type, by design, changes after each communication but the big sum types used when communicating across the unityped channels need to stay the same. Our solution is to record an offset remembering where we currently are in the protocol and thus allowing us to keep injecting values into the initial shared sum types. Our offsets are proven correct using a gadget that can be understood as a cut down version of McBride's one hole contexts [8]. Instead of recording the full path followed by our programs in the finite state machine describing the protocol, we merely record a de-looped version.

Types of the Primitives We include code snippets showing the types of the primitives dealing with communications. Note that Idris 2's funny arrow (`-@`) denotes a linear function space, and consequently all of these primitives are linear in the channel they take as an input (ensuring it cannot be reused after the protocol has been stepped) and systematically return, wrapped in a linear IO monad called `IO1`, a channel indexed by the updated protocol. First, `send` and `recv`, the primitives dealing with sending arbitrary values back and forth. `send` takes an argument of type `ty` while `recv` returns one as the `Res`ult of the communication.

```
send : Channel (Send ty s) e -@ (ty -> IO1 (Channel s e))
recv : Channel (Recv ty s) e -@ IO1 (Res ty (\ _ => Channel s e))
```

Second, `offer` and `select`, the primitives dealing with branching. Both involve a choice in the form of a `Bool` named `b` that is used to compute the next channel state. `offer` gets it from the other thread as a `Res`ult of some communication while `select` requires the caller to pass it and will forward the decision to the other party.

```
offer : Channel (Offer s1 s2) e -@
  IO1 (Res Bool (\ b => ifThenElse b (Channel s1 e) (Channel s2 e)))
select : Channel (Select s1 s2) e -@
  ((b : Bool) -> IO1 (ifThenElse b (Channel s1 e) (Channel s2 e)))
```

Third, `unroll` and `roll`, the primitives dealing with the iso-recursive nature of our (co)inductive protocols. The types of these primitives reveal the role of `Channel`'s second parameter: it is a stack of open session types corresponding to the bodies of the nested fixpoints encountered during execution. `unroll` opens a fixpoint and thus pushes a new entry onto the stack `e` while `roll` does the opposite.

```
unroll : Channel (Fix nm s) e -@ IO1 (Channel s (s :: e))
roll   : Channel s (s :: e) -@ IO1 (Channel (Fix nm s) e)
```

Last but not least, `rec` steps a channel that has reached a recursive call to a fixpoint by looking up the open session and environment associated to the specified `pos`ition in the stack and reinstating them.

```
rec : {pos : _} ->  Channel (Rec pos) e -@
  IO1 (Channel (Fix nm (SessionAt pos e)) (EnvAt pos e))
```

Example Server Omitting the somewhat horrifying types, we include the implementation of the server repeatedly offering to greet the user and execute the adder protocol that we sketched earlier. We use Idris 2's named argument syntax to explicitly pass the name `nm` to the `unroll` and `rec` calls in order to clarify what step of the protocol is being reached.

```
server id ch = do
  ch <- unroll {nm = Nu "ready"} ch
  b # ch <- offer ch
  if b then close ch else do
    let msg = "Hello n°\{show id}!"
    ch <- send ch msg
    ch <- adder 0 ch
    server (1 + id) ch
```

```
adder acc ch = do
  ch <- unroll {nm = Nu "adder"} ch
  b # ch <- offer ch
  if b then do
      ch <- send ch acc
      rec {nm = Nu "ready"} ch
    else do
      (n # ch) <- recv ch
      ch <- rec {nm = Nu "adder"} ch
      adder (acc + n) ch
```

On the left-hand side we are acting as a `server` parametrised by a client `id` number. We first `unroll` the largest fixpoint corresponding to the 'ready' state repeatedly offering the adder protocol, and wait for the client to choose one of the branches we `offer`. If they are done we `close` the channel and terminate, otherwise we `send` a greeting, call the `adder` routine, and eventually continue acting as the server with an incremented id number.

On the right-hand side we are acting as the `adder` parametrised by an accumulator `acc`. We `unroll` the largest fixpoint corresponding to the 'adder' protocol and `offer` the client a choice between getting the running total or sending us more numbers. If they are done, we `send` back the value in the accumulator and, having reached a `Rec` variable in the protocol, use `rec` to jump back to the 'ready' state. Otherwise we `recv` an additional number, add it to the accumulator, use `rec` to jump back to the 'adder' state and continue acting as the `adder`.

Limitations and Future Work

Encoding Uniqueness via Linear Types Unlike Brady's prior work in Idris 1 [2], we are using Idris 2 which does not currently have uniqueness types. We are forced to use the linearity granted to us by Quantitative Type Theory (QTT) [9, 1] to emulate uniquess via a library-wide

invariant. This adds a degree of noise and trust to the implementation. We would ideally want a host system combining both linearity and uniqueness to benefit from enhanced expressivity and efficiency as described by Marshall, Vollmer, and Orchard [7].

Small Runtime Overhead In the current version of our library, we compute a handful of key offset values by induction over the protocol. Correspondingly the protocol is not marked as erased anymore and, if compilation does not specialise the relevant combinators agressively, then we may very well be evaluating these relatively small computations at runtime. As far as we understand, directly adding typed staging to Idris 2 through the use of a two-level type theory à la Kovács [6] would not be enough to solve our issue: for typing purposes, the protocol does need to persist through staging. But it should be possible to move it from QTT's unrestricted to its erased modality.

Ergonomics We found writing the types of inner loops of programs with non-trivial communication patterns to be tedious as they expose quite a lot of the powerful but noisy encoding of syntaxes with binding we use. Figuring out a more user-friendly surface syntax for our protocols is left as future work.

Totality Checking Our lightweight embedding does distinguish smallest and largest fixpoints but Idris 2's productivity checker does not currently take advantage of the knowledge that the user is making steady progress in a potentially infinitely repeating protocol ($\nu x.\ P$) to see that a program is total. Experiments suggest we may be able to introduce a lightweight encoding of a Nakano-style modality [10] for guarded programming to let the user justify that their co-recursive calls are all justified by regular progress.

References

[1] Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018.

[2] Edwin C. Brady. Type-driven development of concurrent communicating systems. *Comput. Sci.*, 18(3), 2017.

[3] Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[4] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[5] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013.

[6] András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6(ICFP):540–569, 2022.

[7] Danielle Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In Ilya Sergey, editor, *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*, volume 13240 of *Lecture Notes in Computer Science*, pages 346–375. Springer, 2022.

[8] Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 287–295. ACM, 2008.

[9] Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016.

[10] Hiroshi Nakano. A modality for recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 255–266. IEEE Computer Society, 2000.

Code Generation via Meta-programming in Dependently Typed Proof Assistants

Mathis Bouverot-Dupuis[1,2] and Yannick Forster[1]

[1] Inria Paris, France
[2] ENS Paris

Abstract

Dependently typed proof assistants offer powerful meta-programming features, which users can take advantage of to implement proof automation or compile-time code generation. This paper surveys meta-programming frameworks in Rocq, Agda, and Lean, with seven implementations of a running example: deriving instances for the `Functor` type class. This example is fairly simple, but sufficiently realistic to highlight recurring difficulties with meta-programming: conceptual limitations of frameworks such as term representation – and in particular binder representation –, meta-language expressiveness, and verifiability as well as current limitations such as API completeness, learning curve, and prover state management, which could in principle be remedied. We conclude with insights regarding features an ideal meta-programming framework should provide. A full experience report in paper form is accessible at https://hal.science/hal-05024207.

All proof assistants support user-extensible tactics and code generation through different meta-programming frameworks. Meta-programs are programs which produce or manipulate other programs as data. They can in particular be used to generate boilerplate code, i.e. code that can be mechanically derived from definitions, thereby increasing the productivity of proof assistant users. Common examples are induction principles [25, 12], equality deciders [25, 10], finiteness proofs [5], countability proofs [5], or substitution functions for syntax [22]. Naturally, the default meta-programming language of a proof assistant is its implementation language, and several proof assistants even come with multiple independent meta-programming frameworks. However, we can observe that meta-programming is not wide-spread on the example of boilerplate generation tools which often fall into one of the following. Either proof assistants come with built-in boilerplate generation support (such as induction principles or type class instances) which is widely used, or tools for generating boilerplate are developed, but not adopted by the community [25, 10, 3]. Lastly, many papers remark that automatic boilerplate generation would be feasible and interesting, but do not carry it out [27, 29, 9, 7]. Furthermore, subcommunities often seem to be split into silos regarding frameworks and we are not aware of scientific comparative work between different frameworks and proof assistants. The notable exception is Dubois de Prisque's PhD thesis [6], using several meta-programming frameworks in Rocq, but not coming with one central example implemented in different frameworks and focusing solely on Rocq.

An additional barrier for adoption is that most frameworks are organically grown and documentation is not accessible to non-experts: pros and cons are often implicitly known by developers but not readily accessible. In fact the situation is so chaotic that, at times, in order to generate boilerplate code authors create ad-hoc meta-programming facilities from scratch [22, 13, 23, 11] instead of taking advantage of existing meta-programming facilities.

On the other hand, the vast choice of meta-programming frameworks also hints that we are at the point where enough evidence is available to evaluate the state of the art and suggest future developments. In this paper, we focus on three major dependently typed proof assistants based on the Calculus of Inductive Constructions (CIC) [4, 17]: Rocq [26], Agda [15, 16], and Lean 4 [14]. We survey their respective meta-programming frameworks: for Rocq we cover OCaml plugins, MetaRocq [20, 21], Ltac2 [19], Elpi [24, 25], and furthermore Agda's Reflection API [28] and Lean 4's meta-programming API.[1] We furthermore explain a novel approach to use Rocq's OCaml API with locally nameless syntax. This means that we focus on systems which are designed as proof assistants with consistent meta-theory, rather than dependently typed programming languages, and focus on those with conceptual similarity and shared foundations. In particular, we do not consider Idris [1, 2], HOL-based systems such as Isabelle, HOL4, or HOL light, or LF-based systems such as Beluga.

We evaluate the different meta-programming frameworks on a simple yet realistic example: automatically deriving instances of the `Functor` type class for a simple family of inductives, covering,

[1] We are not aware of a dedicated publication about meta-programming in Lean 4, but there exists a collaborative book draft [18]. Lean 3's meta-programming was surveyed by Ebner et al. [8].

amongst many other types, options, lists, and trees. For Rocq we e.g. want to generate the following for the list type:

```
Fixpoint fmap {A B : Type} (f : A -> B) (l : list A) : list B :=
  match l with [] => [] | x :: l => f x :: fmap f l end.
```

Many tasks involving automatic boilerplate generation follow the same model as this example: take an inductive as input and produce a term as output. We choose this example because it is simple enough for code to be readable and explainable, yet complex enough to expose issues that arise in more realistic meta-programs, and makes use of common meta-programming features such as type-class search or the ability to extend the global environment.

Paper summary Our paper at https://hal.science/hal-05024207 can be seen as a first step towards a suggested rosetta stone project for meta-programming in Rocq project:

https://github.com/coq-community/metaprogramming-rosetta-stone.

We conclude in the paper that there are conceptual and current limitations to existing frameworks. Conceptually, we observe that:

- *Binder representation* was a recurring issue in the paper,
- *Term quotations* allow to use user syntax directly when constructing and pattern matching on terms, thereby removing the need to spell out fully qualified constant names, implicit arguments, type-class instances, or universe levels,
- *Recursive function generation* is crucial, which became especially apparent for the example in Lean 4, where the need to fall back on primitive recursors prevented us from implementing a plugin with the same features as in the other systems,
- *State* is inherent to meta-programs, which can read and modify the global environment, local environment, and unification state,
- *Verifiability* is a desirable property of frameworks, where ideally formal verification is possible. Verification is not so attractive for users of meta-programs because properties such as well-typedness can be checked a posteriori by the kernel, but implementors of meta-programs might be interested in (partial) correctness guarantees. Indeed, formal specifications can partly replace documentation – which is lacking for all considered frameworks anyways – and can help in writing correct meta-programs, which is far from an easy task considering the complexity of the underlying systems.

Currently, we observe that:

- *The learning curve* of a meta-programming framework is crucial, and writing meta-programs in a different language than that of the underlying proof assistant leads to a steeper curve.
- A proper meta-programming language benefits from adequate tooling, such as a language server, a documentation generator, a package manager, an optimising compiler or an efficient interpreter, and a good integration with the proof assistant's build system.
- Precise performance considerations are outside the scope of this paper, although we do comment on performance when relevant.

Overall, two meta-programming approaches emerge: meta-programming directly in the proof assistant, or using a domain specific language (DSL).

The first option allows the possibility of verifying meta-programs, which is relevant for authors of meta-programs. Verifying meta-programs requires both a specification of the basic meta-programming operations provided by the framework, and adequate means to use these basic specifications in order to derive guarantees about complex meta-programs. The second option does not allow certifying meta-programs, but enables using domain-specific programming language features.

In both cases, one needs a feature-complete meta-programming API, which stays up to date with the evolution of the proof assistant. As implementors of a meta-programming framework, *bootstrapping* the proof assistant gives a feature-complete API for free, but requires significant work a priori (for instance Lean 4's elaborator is bootstrapped). An alternate approach, which MetaRocq and Agda follow, is to do meta-programming directly in the proof assistant, without bootstrapping. Interfacing with the elaborator and kernel is done using a meta-programming monad, which from a user's point of view is very similar to bootstrapping, but requires more work from the implementors.

References

[1] Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

[2] Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2021.9, doi:10.4230/LIPICS.ECOOP.2021.9.

[3] Tej Chajed. Record updates in coq. In *CoqPL 2021: The Seventh International Workshop on Coq for Programming Languages*, 2021. Extended Abstract. URL: https://popl21.sigplan.org/details/CoqPL-2021-papers/3/Record-Updates-in-Coq.

[4] Thierry Coquand and Gérard P Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

[5] Arthur Azevedo de Amorim. Deriving instances with dependent types. In *Proceedings of the Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020. URL: https://popl20.sigplan.org/details/CoqPL-2020-papers/1/Deriving-Instances-with-Dependent-Types.

[6] Louise Dubois de Prisque. *Prétraitement compositionnel en Coq. (Compositional preprocessing in Coq)*. PhD thesis, University of Paris-Saclay, France, 2024. URL: https://tel.archives-ouvertes.fr/tel-04696909.

[7] Catherine Dubois, Nicolas Magaud, and Alain Giorgetti. Pragmatic isomorphism proofs between coq representations: Application to lambda-term families. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France*, volume 269 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.TYPES.2022.11, doi:10.4230/LIPICS.TYPES.2022.11.

[8] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110278.

[9] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. Pi-ware: Hardware description and verification in agda. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 9:1–9:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. URL: https://doi.org/10.4230/LIPIcs.TYPES.2015.9, doi:10.4230/LIPICS.TYPES.2015.9.

[10] Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin's data types with Coq-Elpi. In *CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, pages 167–181, Boston MA USA, France, January 2023. ACM. URL: https://inria.hal.science/hal-03800154, doi:10.1145/3573105.3575683.

[11] Jason Gross, Théo Zimmermann, Miraya Poddar-Agrawal, and Adam Chlipala. Automatic test-case reduction in proof assistants: A case study in coq. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 18:1–18:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ITP.2022.18, doi:10.4230/LIPICS.ITP.2022.18.

[12] Bohdan Liesnikov, Marcel Ullrich, and Yannick Forster. Generating induction principles and subterm relations for inductive types using metacoq. *CoRR*, abs/2006.15135, 2020. URL: https://arxiv.org/abs/2006.15135, arXiv:2006.15135.

[13] Nicolas Magaud. Towards automatic transformations of coq proof scripts. In Pedro Quaresma and Zoltán Kovács, editors, *Proceedings 14th International Conference on Automated Deduction in Geometry, ADG 2023, Belgrade, Serbia, 20-22th September 2023*, volume 398 of *EPTCS*, pages 4–10, 2023. doi:10.4204/EPTCS.398.4.

[14] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Lecture Notes in Computer Science*, pages 625–635. 2021. doi:10.1007/978-3-030-79876-5_37.

[15] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[16] Ulf Norell, Nils Anders Danielsson, Jesper Cockx, and Andreas Abel. Agda wiki. http://wiki.portal.

chalmers.se/agda/pmwiki.php.

[17] Christine Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993. doi:10.1007/BFb0037116.

[18] Arthur Paulino, D Testa, E Ayers, H Böving, J Limperg, S Gadgil, and S Bhat. Metaprogramming in lean 4. *Online Book. https://github. com/arthurpaulino/lean4-metaprogramming-book*, 2024.

[19] Pierre-Marie Pédrot. Ltac2: Tactical warfare. In *The 5th International Workshop on Coq for Programming Languages (CoqPL 2019)*, 2019. Talk at CoqPL 2019, affiliated with POPL 2019. URL: https://popl19.sigplan.org/details/CoqPL-2019/8/Ltac2-Tactical-Warfare.

[20] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020. URL: https://inria.hal.science/hal-02167423, doi:10.1007/s10817-019-09540-0.

[21] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371076.

[22] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

[23] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk. Proviola: A tool for proof reanimation. *CoRR*, abs/1005.2672, 2010. URL: http://arxiv.org/abs/1005.2672, arXiv:1005.2672.

[24] Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). In *The Fourth International Workshop on Coq for Programming Languages*, Los Angeles (CA), United States, January 2018. URL: https://inria.hal.science/hal-01637063.

[25] Enrico Tassi. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2019.29, doi:10.4230/LIPIcs.ITP.2019.29.

[26] The Coq Development Team. The coq proof assistant, September 2024. doi:10.5281/zenodo.14542673.

[27] Cas van der Rest and Wouter Swierstra. A completely unique account of enumeration. *Proc. ACM Program. Lang.*, 6(ICFP):411–437, 2022. doi:10.1145/3547636.

[28] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In *International Symposium on Implementation and Application of Functional Languages*, 2012. URL: https://api.semanticscholar.org/CorpusID:18658569.

[29] Marcell van Geest and Wouter Swierstra. Generic packet descriptions: verified parsing and pretty printing of low-level data. In Sam Lindley and Brent A. Yorgey, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, pages 30–40. ACM, 2017. doi:10.1145/3122975.3122979.

Towards Being Positively Negative about Dependent Types

Jan de Muijnck-Hughes

University of Strathclyde, UK
Jan.de-Muijnck-Hughes@strath.ac.uk

Negativity is Important for Proving and Programming Dependently typed programming languages, such as Idris2 [2] and Agda [3], provide expressive environments in which we can reason about and *run* our programs. Consider, for example, the following Idris2 definition of natural numbers (`Nat`) and two predicates (`NonZero` & `IsZero`) that state if a given natural number is non-zero or not:

```
data Nat = Z | S Nat     data NonZero : Nat -> Type     data IsZero : Nat -> Type
                           where NZ : NonZero (S n)       where IZ : IsZero Z
```

Decision procedures support reasoning about the correctness of `NonZero` and `IsZero`, by evidencing the construction of valid instances of the predicates or supplying a proof of falsity if not. These proofs are then repurposed as verified functions for programming. For example, consider the following 'proof' (`nonZero`) that `NonZero` is correct:

```
nonZero : (n : Nat) -> Dec (NonZero n)
nonZero Z = No absurd
  where
    absurd : NonZero Z -> Void
    absurd NZ impossible
nonZero (S x) = Yes NZ
```

```
data Dec : a -> Type where
  Yes : (prf : a) -> Dec a
  No  : (contra : a -> Void) -> Dec a
```

Our proof (`NonZero`) will produce an inhabitant of `NonZero` if the input *is* non-zero (`NZ`), or a proof of void if not (`absurd`). The generic datatype `Dec` encapsulates the result of `NonZero`, where `Yes` represents the positive result and `No` the negative result. If we wish, however, to use the negative result of `nonZero` at runtime then we cannot do so and determine why the function failed. Although it is self-evident that a negative result for `nonZero` implies that the input was non-zero, more complex predicates can fail for a variety of reasons.

Runtimes are Positive; Negativity is not Generally speaking, predicates are *positive* pieces of information and construction of decision procedures using `Dec` only enables runtime evidence to be produced when the 'happy' (positive) path is taken. All traces of *negative* information (i.e. falsity) are now a (careless) compile-time whisper. If we are *theorem proving* in dependently-typed languages, runtime reports of failure are not important; our code is proven correct. If we are *programming* in a dependently-typed language then knowing why a decision procedure failed is important; errors need to be reported. If we start to think more positively about being negative, we can start to report negative information more positively.

Being Constructive about Negation is a Positive The MSFP 2022 talk 'Data Types with Negation' discussed the idea of using *Constructive Negation* to rethink how we can work positively with negative information when programming [1]. Within dependently typed languages we can exploit constructive negation to rethink how we represent decidable decisions. For instance, consider the following *positive* definitions:

```
record Decidable where
  constructor D
  Positive : Type
  Negative : Type
  0 Cancelled : Positive -> Negative -> Void
```

```
0
Dec : Decidable -> Type
Dec (D positive negative no)
  = Either negative positive
```

`Decidable` is a datatype encapsulating the runtime irrelevant proof (identified by the `0` quantifier) that two positive pieces of information cancel each other out. Instances of `Decidable` are the input to a positive `Dec` which is translated to an instance of `Either`, where the positive truth is made `Right` and the positive evidence of falsity is `Left`. With `Decidable` and `Dec`, are decision procedures are now positive. We can illustrate our positive decisions by pairing our definitions of `IsZero` and `NonZero` together to produce `ISZERO`:

```
ISZERO : Nat -> Decidable
ISZERO n = D (IsZero n) (IsSucc n) prf
  where prf : IsZero n -> NonZero n -> Void
        prf IZ NZ impossible
```

```
isZero : (n : Nat) -> Dec (ISZERO n)
isZero Z     = Right IZ
isZero (S k) = Left  NZ
```

The function `prf` evidences that a number cannot be simultaneously zero and non-zero: our proof of falsity. The function `ISZERO` constructs the decidable predicate for a given natural number, and `isZero` is proof that we can decide if a number is non-zero or not. We can even reuse the same predicates, albeit flipped, to produce a decision procedure for `NONZERO`:

```
NONZERO : Nat -> Decidable
NONZERO n = D (IsSucc n) (IsZero n) prf
  where prf : NonZero n -> IsZero n -> Void
        prf NZ IZ impossible
```

```
nonZero : (n : Nat) -> Dec (NONZERO n)
nonZero Z     = Left  IZ
nonZero (S k) = Right NZ
```

Being Positive is Hard Through careful selection of our datatypes, we can develop the foundations of a 'positive' library for decision procedures. If we are not careful with our constructions we can easily end up with incorrect decisions being made. Take for example, quantifying over lists using 'All' and 'Any' predicates. The opposite of the All quantifier is the Any quantifier. Either all elements satisfy the positive predicate, or we will traverse the list until a positive instance of the negative predicate is found. The opposite of `Any` is, unfortunately, not a simple swapping of predicates as we saw with `NONZERO` and `ISZERO`. For the `Any` quantifier, we need to reverse the polarity of the predicates as well. A positive `Any` requires us to traverse the list and build the negative predicate until we find our positive one. The opposite of `Any` is that all items in the list produced negative predicates.

This Talk is about Positively Negative Programming In this talk, I will report *work-in-progress* that explores what it means to be *positively negative* when programming with dependent types. I will report how constructive negation re-frames not only the construction of a library[1] of positive decision procedures, but my experience using these procedures when programming. Specifically, I will report on reasoning about standard datatypes (natural numbers, lists, pairs, and strings) including their decidable equality, as well as their use when elaborating concrete syntax to intrinsically-typed terms for the Simply-Typed Lambda Calculus. Finally, I will examine how being so positive in one's negativity can reshape existing approaches to dependently-typed programming and report our decision procedures negativity more positively.

[1] https://github.com/jfdm/positively-negative

References

[1] Robert Atkey. 'Data Types with Negation'. Extended Abstract (Talk Only) at Ninth Workshop on Mathematically Structured Functional Programming, Munich, Germany, 2nd April 2022. 2022. URL: https://youtu.be/mZZjOKWCF4A.

[2] Edwin C. Brady. 'Idris 2: Quantitative Type Theory in Practice'. In: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*. 2021, 9:1–9:26. DOI: 10.4230/LIPIcs.ECOOP.2021.9.

[3] The Agda Development Team. *Agda*. 2023. URL: https://github.com/agda/agda.

Implementing a Typechecker for an Esoteric Language: Experiences, Challenges, and Lessons

Alex Rice

University of Edinburgh, UK
alex.rice@ed.ac.uk

The type theory CATT [FM17] is a dependently typed language which models weak ∞-categories [Ben20]. Recent work has introduced $\text{CATT}_{\mathsf{su}}$ [FRVR22] and $\text{CATT}_{\mathsf{sua}}$ [FRV24], which extend CATT with a non-trivial definitional equality relation and respectively model strictly unital ∞-categories and strictly unital and associative ∞-categories. Normalisation algorithms for both of these theories are given by demonstrating the existence of a reduction system which is both strongly terminating and confluent. Naively implementing a typechecker based on such a system can be inefficient, due to the overhead of generating each intermediate term produced by these reductions.

The reductions found in these systems are non-standard, with neither theory containing lambda abstraction, application, or beta reduction, preventing techniques from existing literature (e.g. [Abe13, GSB19, GSA+22, AHS95]) from being directly applicable. These reductions are also syntactically complex, and have non-trivial interactions, which motivates further study of normalisation procedures for these theories.

In the talk I will introduce an implementation of CATT, $\text{CATT}_{\mathsf{su}}$, and $\text{CATT}_{\mathsf{sua}}$, which can be found at:

https://github.com/alexarice/catt-strict

Instead of delving too heavily on the specifics of the languages being typechecked, the talk will instead focus on the overall structure of the tool, some of the decisions made in its construction, and more generally my experience and the lessons learnt on the way.

When implementing a typechecker, there are multiple fundamental design decisions with different tradeoffs. The chosen design was optimised to achieve the following objectives, which don't necessarily fall within the core remit of a typechecker or interpreter.

- Type Inference: type information can be omitted in places where it can be inferred.
- Efficient evaluation: by utilising Normalisation by Evaluation (NbE) [Abe13, BSV91], evaluation of larger terms is near instantaneous.
- Preservation of variable names: variable names are often chosen by the user to convey content, but named variables are problematic to deal with due to alpha equivalence. Our tool utilises de Bruijn levels to represent variables but stores the original variable names in the context.
- Top-level bindings: Terms can be bound to global symbols. Furthermore, the tool is careful not to eagerly replace a top-level symbol by its definition, as similarly to the above point, the name of a top level symbol often conveys the intent of the user, making the term easier to read.
- Detailed error reporting: The majority of a user's interaction with a typechecker will likely be receiving error messages while debugging. The errors in our tool take the following form:

```
Error: Given term "x" does not match inferred term "y"
    ┌─[examples/test.catt:19:34]
    │
 19 │ def error_test x{f}y = comp (x{f}x)
    │                                  ┬
    │                                  └── Given term
────┘
```

Displaying such errors requires a correspondence between parsed terms in the memory of the tool and their location in the text.

While these problems, in particular the first two, have been individually studied, figuring out how to combine these into a single tool can be difficult. This is amplified when working on a non-standard language, as much of the existing literature focuses on the lambda calculus and its extensions.

The core of the typechecker is based on a bidirectional typechecking algorithm (see [DK21]), which splits the various typing rules into inference rules and checking rules. Contrary to more familiar type theories, all types in Catt are inferable, and we instead find a split between rules where the *context* can be inferred or checked.

The form of NbE implemented in the tool is largely inspired by the paper "Implementing a modal dependent type theory" [GSB19], although we note that the theory Catt is vastly different to the type theory they present.

The tool contains 3 main representations of syntax:

- Raw syntax: the raw syntax is intended to match the written syntax of the language as closely as possible, and represents variables by their names. Its primary purpose is to be the target of parsing and the source of printing; terms are only displayed by converting them to raw syntax first.

  The data structure implementing the raw syntax has a generic type parameter which can be used to add optional annotations to each constructor. We instantiate this type parameter to a type of "text locations" when the syntax arises from parsing and can otherwise instantiate it to the unit type. These (optional) text locations enable more informative error information.

- Core syntax: this syntax represents typechecked terms, and uses de Bruijn levels. Our typechecking procedure takes raw syntax and produces core syntax.

- Normal form syntax: this represents the possible normal forms of evaluation within the theory. The NbE evaluation algorithm converts a core syntax term to a normal form syntax term, and these terms can be quoted back to core syntax terms.

In practice, the use of all three types of syntax are heavily intertwined, due to the nature of dependent typing. During the talk, I will explain how this split simplifies various stages of the typechecking algorithm.

I will also explain what I believe is the largest flaw of this setup, the difficulty of debugging; core and normal form syntax is hard to print in an informative way, and implementation errors were often not caught early, making it difficult to implement the non-trivial reductions of these theories.

References

[Abe13] Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity*. Habilitation thesis, Ludwig-Maximilians-Universität München, 2013.

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, volume 953, pages 182–199, 1995.

[Ben20] Thibaut Benjamin. *A Type Theoretic Approach to Weak Omega-Categories and Related Higher Structures*. Theses, Institut Polytechnique de Paris, November 2020.

[BSV91] Ulrich Berger, Helmut Schwichtenberg, and R. Vermuri. An inverse of the evaluation functional for typed Lambda-calculus. In R. Vermuri, editor, *6th Annual IEE Symposium on Logic in Computer Science (LICS'91)*, pages 203–211, Amsterdam, 1991. Ludwig-Maximilians-Universität München.

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional Typing. *ACM Comput. Surv.*, 54(5):98:1–98:38, May 2021.

[FM17] Eric Finster and Samuel Mimram. A type-theoretical definition of weak ω-categories. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, June 2017.

[FRV24] Eric Finster, Alex Rice, and Jamie Vicary. A Syntax for Strictly Associative and Unital ∞-Categories. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '24, pages 1–13, New York, NY, USA, July 2024. Association for Computing Machinery.

[FRVR22] Eric Finster, David Reutter, Jamie Vicary, and Alex Rice. A Type Theory for Strictly Unital ∞-Categories. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, pages 1–12, New York, NY, USA, August 2022. Association for Computing Machinery.

[GSA+22] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. Controlling unfolding in type theory, October 2022.

[GSB19] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *blott: An Implementation of a Modal Dependent Type Theory*, 3(ICFP):107:1–107:29, July 2019.

A Two-level Foundation for the Calculus of Constructions

Pietro Sabelli[1]
j.w.w. Maria Emilia Maietti[2]

[1] Department of Logic, Institute of Philosophy of the Czech Academy of Sciences
sabelli@flu.cas.cz
[2] Department of Mathematics, University of Padova
maietti@math.unipd.it

As a foundation for mathematics, type theory is affected by a long-dated and pervasive tension between intensional and extensional representations of mathematical entities. Intuitively, mathematicians want type theories able to consider mathematical objects independently of their particular presentations, while computer scientists are more bound to these when looking for theories with good computational properties. This dichotomy has practical consequences in the proof-assistant world, with many relying on the easier-to-implement intensional theories (such as Agda, Coq, or Lean), and fewer implementing full extensional theories supporting the reflection rule (such as Nuprl or the more recent Andromeda).

In [MS05], a way out of the intensional vs. extensional impasse has been advocated through the notion of *two-level foundation*[1]. According to this paradigm, a type-theoretic foundation for mathematics should consist of two distinct type theories, each assigned with a precise role in agreement with its nature: an extensional type theory as the actual system for the foundations of mathematics, and an intensional type theory as a functional programming language in which to implement the former through a suitable interpretation. The problem of modeling full extensional type theories using intensional ones has been notably addressed in the case of Martin-Löf's type theory by Hofmann in [Hof95] in the search of conservativity results, which led to add some specific axioms to the considered intensional theory.

Always in [MS05], the authors conceived the Minimalist Foundation, a two-level foundation which can be considered either as a predicative version of the Calculus of (Inductive) Constructions [CH88, PM15], or as a version of Martin-Löf's type theory enriched with a primitive notion of proposition. The Minimalist Foundation has then been fully formalized in [Mai09] as consisting of: an extensional level **emTT** (for *extensional minimal type theory*) which extends the extensional version of Martin-Löf's in [Mar84] in particular with a power constructor, quotient sets, and proof-irrelevance; an intensional level **mTT** (for *minimal type theory*) which extends the intensional version of Martin-Löf's in [NPS90]; and an interpretation of the former in a setoid model built on the latter. The Minimalist Foundation was introduced to serve as a common-core foundation in which to develop mathematics agnostically, since it can be interpreted in many of the most relevant set-theoretic and type-theoretic foundations for mathematics; in particular, both its levels are interpretable in Homotopy Type Theory [CM22].

Here, we extend a fragment of the Calculus of Inductive Constructions $\mathbf{CC}_{\mathsf{ML}}$ supporting the basic inductive types of Martin-Löf to a two-level foundation, by taking the impredicative version $\mathbf{emTT}^{\mathsf{imp}}$ of **emTT** as its extensional level. More in detail, the theory $\mathbf{emTT}^{\mathsf{imp}}$ is obtained by adding to **emTT** an impredicative universe of propositions quotiented by logical equivalence or, equivalently, by adding a powerset constructor; we claim that such a theory pro-

[1] This has not to be confused with the notion of *two-level type theory* 2LTT [ACKS23], introduced later for a different purpose.

vides the internal language of Penon's quasitoposes and show that the results for the Minimalist Foundation proved in [Mai09] and [MS24] can be extended to it.

Firstly, we prove that $\mathbf{emTT}^{\mathsf{imp}}$ and $\mathbf{CC}_{\mathsf{ML}}$ are mutually interpretable. The main obstacle comes from the fact that, whilst resembling closely the two-level structure of Martin-Löf's type theory, where the extensional level is obtained as an extension of the intensional one with the addition of an equality reflection rule, the presence of a universe of propositions *quotiented by logical equivalence* does not make $\mathbf{emTT}^{\mathsf{imp}}$ a direct extension of $\mathbf{CC}_{\mathsf{ML}}$. Whether $\mathbf{CC}_{\mathsf{ML}}$ can be interpreted into $\mathbf{emTT}^{\mathsf{imp}}$ is therefore not trivial. We will answer it positively using the canonical isomorphisms technique applied to a bridge theory obtained by extending $\mathbf{emTT}^{\mathsf{imp}}$ with the axiom of propositional extensionality. As a byproduct, we will also conclude that the assumption $\mathsf{propext}$ of propositional extensionality is conservative over $\mathbf{emTT}^{\mathsf{imp}}$, in the sense that any proposition expressible in $\mathbf{emTT}^{\mathsf{imp}}$ and provably true in $\mathbf{emTT}^{\mathsf{imp}} + \mathsf{propext}$ is already true in $\mathbf{emTT}^{\mathsf{imp}}$.

Secondly, we prove that $\mathbf{emTT}^{\mathsf{imp}}$ is equiconsistent with its classical version through a double-negation translation. Thanks to the possibility of doing impredicative encodings in $\mathbf{emTT}^{\mathsf{imp}}$, this result includes also the translation of inductive and coinductive predicates, and therefore, according to the results in [MS23, Sab24], of (co)inductively generated formal topologies; in the predicative setting of **emTT** this is still an open problem; likewise, it is an open problem to determine if such double-negation translation can be extended to support the (co)inductive schemes of the full Calculus of Inductive Constructions.

Finally, we show how to extend the results in [Mai05] to prove that $\mathbf{emTT}^{\mathsf{imp}}$ provides the internal language of quasitoposes. The only notable change with respect to the cited work is that logic in a quasitopos must be interpreted through *strong* monomorphisms, which only in a genuine topos coincide with all monomorphisms.

References

[ACKS23] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *Mathematical Structures in Computer Science*, 33(8):688–743, 2023.

[CH88] T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76:95–120, 1988.

[CM22] M. Contente and M. E. Maietti. The compatibility of the minimalist foundation with homotopy type theory. 2022.

[Hof95] M. Hofmann. Conservativity of equality reflection over intensional type theory. In *International Workshop on Types for Proofs and Programs*, pages 153–164. Springer, 1995.

[Mai05] M. E. Maietti. Modular correspondence between dependent type theories and categories including pretopoi and topoi. *Math. Structures Comput. Sci.*, 15(6):1089–1149, 2005.

[Mai09] M. E. Maietti. A minimalist two-level foundation for constructive mathematics. *Ann. Pure Appl. Logic*, 160(3):319–354, 2009.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory, notes by G. Sambin of a series of lectures given in Padua, June 1980.* Bibliopolis, Naples, 1984.

[MS05] M. E. Maietti and G. Sambin. Toward a minimalist foundation for constructive mathematics. In *From sets and types to topology and analysis*, volume 48 of *Oxford Logic Guides*, pages 91–114. Oxford Univ. Press, Oxford, 2005.

[MS23] M. E. Maietti and P. Sabelli. A topological counterpart of well-founded trees in dependent type theory. *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 3 - Proceedings of MFPS XXXIX , November 2023.

[MS24] M. E. Maietti and P. Sabelli. Equiconsistency of the minimalist foundation with its classical version. *Annals of Pure and Applied Logic*, page 103524, 2024.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin Löf's Type Theory.* Clarendon Press, Oxford, 1990.

[PM15] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[Sab24] P. Sabelli. A topological reading of inductive and coinductive definitions in dependent type theory, 2024.

A Timed Predicate Temporal Logic Sequent Calculus

Javier Enríquez Mendoza[1], Sam Speight[1], and Vincent Rahli[1]

[1]University of Birmingham, UK

Abstract

Our society strongly depends on critical information infrastructures such as autonomous vehicles, blockchain applications, IoT infrastructures, etc. Because of the complexity of these systems, guaranteeing that they operate in a correct and timely fashion is hard to achieve. In this abstract, we present a logical framework that allows reasoning about timing properties of such systems, and demonstrate its applicability through a series of examples.

Temporal Logic Temporal logic is used to describe and reason about the behavior of systems over time. A prominent example of such a logic is Timed Propositional Temporal Logic (TPTL) [4, 1, 2], an extension of Linear Temporal Logic (LTL) [6] with explicit clock variables that facilitates reasoning about time-bound properties. TPTL has notably been extended with past operators (TPTL + Past) in [5], which allows for reasoning about both past and future events, as well as quantifiers in [1]. Other "timed" temporal logics have been proposed such as Timed CTL [3, Sec.9.2], which most notably extends CTL with clocks and constraints on clocks, and for which model checking algorithms exist. We focus here on designing a calculus for a variant of TPTL + Past with quantifiers, which is fully formalised in Agda.

The syntax of TPTL is as follows, where $\mathbf{U}$ and $\bigcirc$ are the usual "until" and "next" LTL operators; p is an atomic proposition; x is a clock variable; $c \in \mathbb{Q}$; $\sim \in \{\leq, <, =, >, \geq\}$ allows comparing two time expressions; and the "freeze" formula $x \cdot \varphi$ binds the current time to x in φ:

$$\varphi \quad ::= \quad p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi \,\mathbf{U}\, \varphi \mid \bigcirc\varphi \mid x \sim c \mid x \cdot \varphi$$

Temporal operators such as "eventually" and "always" are defined as usual: $\Diamond\varphi := \mathtt{true}\,\mathbf{U}\,\varphi$ and $\Box\varphi := \neg\Diamond\neg\varphi$. Using the "freeze" operator, one can then capture that the formula φ eventually occurs by "time" c: $x \cdot \Diamond(x \leq c \wedge \varphi)$. To understand this, let us consider the semantics of a subset of TPTL, given in terms of a mapping $\rho = (\sigma, \tau)$ from $\mathbb{N}$ to states, where a state is the pair of a valuation on atoms and a timestamp, a $i \in \mathbb{N}$, and a variable valuation v:

$$\begin{array}{lcl} \rho, i, v \models \varphi \,\mathbf{U}\, \psi & \iff & \exists j > i.(\rho, j, v \models \psi) \wedge \forall k \in [i, j).(\rho, k, v \models \varphi) \\ \rho, i, v \models x \sim c & \iff & \tau_i - v(x) \sim c \\ \rho, i, v \models x \cdot \varphi & \iff & \rho, i, v[x \mapsto \tau_i] \models \varphi \end{array}$$

The semantics of $x \cdot \Diamond(x \leq c \wedge \varphi)$ w.r.t. ρ, i, v states that there exists a time τ_j at which φ is true, for some $j > i$ such that $\tau_j \leq \tau_i + c$.

Our Calculus We now present an extension of TPTL + Past, with quantifiers, a more general comparison operator, a "discrete" semantics, and a sequent calculus.

The syntax of this calculus is as follows (some operators are omitted for space reasons), where $\mathbf{S}$ and $\mathbf{Y}$ are the "since" and "yesterday" past counterparts of $\mathbf{U}$ and $\bigcirc$; the **Data** type is used to capture the information exchanged between agents; i ranges over a set of agents Agent; d ranges over a set of data Data; A ranges over a set of sets of agents Agents; an atom a is either an atomic proposition p, or a "send" atom $\mathtt{send}(i, d, A)$ stating that agent i sent the data d to the set of agents A, or a "receive" atom $\mathtt{recv}(i, d, j)$ stating that agent i received the data d from agent j, or an "internal event" atom $\mathtt{inter}(i, d)$ stating that d happened at agent i. It can be observed that the set of comparison operators only includes strict less-than

and equality, unlike TPTL's operators. This is because the remaining operators can be defined using these two along with conjunction and disjunction, thanks to the fact that our comparison operators are not dependent on the current time.

$$\begin{array}{rcl} \varphi & ::= & a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \bigcirc\varphi \mid \varphi\,\mathbf{S}\,\varphi \mid \mathbf{Y}\,\varphi \mid t \sim t \mid x \cdot \varphi \mid \forall u : T.\varphi \mid \exists u : T.\varphi \\ T & ::= & \mathbf{Agent} \mid \mathbf{Agents} \mid \mathbf{Data} \qquad \sim \; ::= \; < \mid = \\ t & ::= & x \mid \mathtt{0} \mid t \bullet t \mid \mathtt{s}(t) \qquad a \; ::= \; p \mid \mathtt{send}(i,d,A) \mid \mathtt{recv}(i,d,j) \mid \mathtt{inter}(i,d) \end{array}$$

Let us now adapt and extend TPTL's semantics to our calculus. There, formulas are interpreted w.r.t.: (1) a timestamp t; (2) a function r from timestamps to states, called a run; (3) an interpretation function π, which given a state and an atom, captures whether the atom is true in that state; and (4) a variable valuation (we only present a subset for space reasons):

$$\begin{array}{lcl} \pi, r, t, v \models a & \iff & \pi(r(t))(a) \\ \pi, r, t, v \models \varphi\,\mathbf{S}\,\psi & \iff & \exists t' < t.(\pi, r, t', v \models \psi) \wedge \forall t'' \in (t', t].(\pi, r, t'', v \models \varphi) \\ \pi, r, t, v \models \mathbf{Y}\,\varphi & \iff & t > 0 \text{ and } \pi, r, t-1, v \models \varphi \\ \pi, r, t, v \models t_1 \sim t_2 & \iff & [\![t_1]\!]_v \sim [\![t_2]\!]_v \\ \pi, r, t, v \models x \cdot \varphi & \iff & \pi, r, t, v[x \mapsto t] \models \varphi \\ \pi, r, t, v \models \forall u : T.\varphi & \iff & \pi, r, t, v[u \mapsto z] \models \varphi \text{ for all } z \in [\![T]\!] \end{array}$$

$$\begin{array}{lcl} [\![x]\!]_v & := & v(x) \\ [\![\mathtt{0}]\!]_v & := & 0 \\ [\![t_1 \bullet t_2]\!]_v & := & [\![t_1]\!]_v + [\![t_2]\!]_v \\ [\![\mathtt{s}(t)]\!]_v & := & [\![t]\!]_v + 1 \end{array} \qquad \begin{array}{lcl} [\![\mathbf{Agent}]\!] & := & \mathsf{Agent} \\ [\![\mathbf{Agents}]\!] & := & \mathsf{Agents} \\ [\![\mathbf{Data}]\!] & := & \mathsf{Data} \end{array}$$

Thanks to the more general comparison operator, a time bounded version of the $\Diamond$ operator can now be defined in a more straightforward way: $\Diamond_t\varphi := x \cdot \Diamond(y \cdot y \leq x \bullet t \wedge \varphi)$, stating that φ happens sometime in the future by "time" t (where $t_1 \leq t_2$ stands for $t_1 < t_2 \vee t_1 = t_2$).

Several proof systems have been proposed for LTL-like logics. However, the until operator is known to pose difficulties. A labeled Natural Deduction system was proposed in [7] to circumvent those difficulties. Simplifying and generalizing that system, we propose a labeled sequent calculus that allows reasoning about until and timing properties using the same machinery. We show below the until rules, which illustrate key aspects of our calculus:

$$\frac{\Gamma \vdash_{\mathtt{t}} \mathtt{t} < \mathtt{t}_1 \qquad \Gamma \vdash_{\mathtt{t}_1} \psi \qquad \Gamma, \mathtt{t} \leq x, x < \mathtt{t}_1 \vdash_x \varphi}{\Gamma \vdash_{\mathtt{t}} \varphi\,\mathbf{U}\,\psi}\;\mathbf{U}\,\mathrm{R} \qquad \frac{\Gamma, \mathtt{t} < x, (\psi)^x, (\varphi)^{[\mathtt{t},x)} \vdash_{t_1} \gamma}{\Gamma, (\varphi\,\mathbf{U}\,\psi)^{\mathtt{t}} \vdash_{t_1} \gamma}\;\mathbf{U}\,\mathrm{L}$$

A sequent is of the form $\Gamma \vdash_{\mathtt{t}} \varphi$, stating that φ holds at time $\mathtt{t}$ in the context Γ. An hypothesis is of the form $(\psi)^r$, where r is a time annotation, which can either be: (1) a time expression $\mathtt{t}$, (2) a time interval such as $[t_1, t_2]$ or $[t_1, t_2)$, or (3) an "empty" annotation stating that the hypothesis holds at all times. The $\mathbf{U}\,\mathrm{R}$ rule states that for $\varphi\,\mathbf{U}\,\psi$ to be true at time t it must be that there exists a time t_1 greater than t at which ψ holds, and that φ holds between t and t_1. The $\mathbf{U}\,\mathrm{L}$ rule states that if $\varphi\,\mathbf{U}\,\psi$ is true at time t then there must be a time x (a variable) greater than t at which φ holds, and φ must be true between t and x.

Let us illustrate this calculus through a straightforward example where all nodes are correct and communication is synchronous, ensuring that all sent messages are received within a time interval T, captured by the following formula (if an agent sends a message d at time t to a group, all members of that group will receive the message by time $t + T$):

$$\forall a : \mathbf{Agent}.\forall d : \mathbf{Data}.\forall A : \mathbf{Agents}.\forall b : \mathbf{Agent}.\Box(\mathtt{send}(a,d,A) \rightarrow \Diamond_T(b \in A \rightarrow \mathtt{recv}(a,d,b)))$$

Additionally, we assume that any information received by a node is immediately shared with other node (whenever an agent b receives a message, it immediately sends it to the set of nodes containing only c):

$$\forall a : \mathbf{Agent}.\forall d : \mathbf{Data}.\forall b : \mathbf{Agent}.\forall c : \mathbf{Agent}.\Box(\mathtt{recv}(a, d, b) \rightarrow \mathtt{send}(b, d, \{c\}))$$

Using our calculus, from the above assumptions, one can then derive the following formula, stating that if a node a sends a message d to node b, then c will also receive d before $2T$:

$$\forall a : \mathbf{Agent}.\forall d : \mathbf{Data}.\forall b : \mathbf{Agent}.\forall c : \mathbf{Agent}.\mathtt{send}(a, d, \{b\}) \rightarrow \Diamond_{2T}(\mathtt{recv}(b, d, c))$$

Based on this calculus, we now plan on designing a type system tailored to the unique demands of distributed systems, aligning type-based constraints with the above logical framework.

References

[1] Rajeev Alur and Thomas A. Henzinger. Real-time logics: complexity and expressiveness. In *LICS*, pages 390–401, June 1990. URL: https://ieeexplore.ieee.org/document/113764/?arnumber=113764, doi:10.1109/LICS.1990.113764.

[2] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–203, January 1994. doi:10.1145/174644.174651.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[4] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of tptl and mtl. In *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS '05, page 432–443, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11590156_35.

[5] Laura Bozzelli, Aniello Murano, and Loredana Sorrentino. Alternating-time temporal logics with linear past. *Theoretical Computer Science*, 813:199–217, April 2020. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397519307546, doi:10.1016/j.tcs.2019.11.028.

[6] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977. doi:10.1109/SFCS.1977.32.

[7] Marco Volpe. *Labeled natural deduction for temporal logics*. PhD thesis, University of Verona, Italy, 2010. URL: https://opac.bncf.firenze.sbn.it/bncf-prod/resource?uri=TD11087417.

Verifying Z3 RUP Proofs with the Interactive Theorem Provers Coq/Rocq and Agda

Harry Bryant[1]*, Andrew Lawrence[2],
Monika Seisenberger[1]†, and
Anton Setzer[1]‡

[1] Swansea University, Dept. of Computer Science, Swansea, Wales, UK
{harry.bryant,m.seisenberger,a.g.setzer}@swansea.ac.uk
[2] Siemens Mobility, Chippenham, England, UK andrew.lawrence@siemens.com

Ensuring the correctness of safety-critical systems, such as in railway control systems, is as important as ever. To achieve this, machine-assisted theorem proving is increasingly used in the railway domain. Tools such as Z3 [12] are employed to formally prove that such systems meet the required standards (see e.g. the papers by our group [1, 7]). However, any of these solvers may have flaws or implement optimisations that produce incorrect results. To increase trust, proof checking offers an independent check of the Z3 output. We are developing a verified proof checker for Z3 using its new Reverse Unit Propagation (RUP) format. As a first step, we have focused on propositional formulae in conjunctive normal form (CNF) [17]. The new RUP proof format was introduced to the Z3 theorem prover in September 2022, replacing resolution [2]. Proof checking for other proof formats for SAT and SMT solving has been performed in, e.g., [3, 5, 4, 8, 26, 11, 27, 13, 9].

The notion of a RUP proof was introduced by van Gelder [15, 14] in 2008. It addresses the issue that resolution proofs can be too lengthy to store feasibly while still allowing efficient checking. The underlying concept is proof verification by Goldberg and Novikov [16], where unit propagation checks unsatisfiability without storing full resolution proofs. Van Gelder refined this into RUP, requiring each derived clause to cause a contradiction when added, making proofs more compact and efficient.

RUP takes logical statements written in CNF, where each clause is a disjunction of literals $\{x_1, \ldots, x_n\}$. Negation of a literal x_i simply switches from x_i to $\neg x_i$, or vice versa. A formula is a conjunction of clauses. Z3 deals with formulae not in CNF by translating them using the Tseitin transformation [25, 22, 20].

The goal of a SAT or SMT solver is to decide whether clauses Γ are unsatisfiable, which means Γ entails falsity. Intermediate steps of a proof of unsatisfiability derive from Γ a set of clauses Δ, such that the conjunction of the clauses in Γ entails all clauses in Δ, with the ultimate proof including the empty clause in Δ.

A RUP inference of a clause $C = \{x_1, , \ldots, x_n\}$ from assumptions (clauses) Γ is correct, if from $\Gamma' := \Gamma, \{\neg x_1\}, \ldots, \{\neg x_n\}$ we can derive the empty clause $\{\}$ using unit-clause propagation only. A RUP proof from an initial clause set Γ_0 is a sequence of clauses C_i, for $i \geqslant 1$, such that for all i C_i is a RUP inference from Γ_{i-1}, where $\Gamma_j = \Gamma_{j-1} \cup \{C_j\}$, for $j \geqslant 1$. If some C_j is the empty clause $\{\}$, the sequence is called a RUP refutation [15]. Checking a RUP inference is done as follows: Divide Γ' into non-unit clauses Γ_{nunit} (clauses of length $\geqslant 2$) and unit clauses Γ_{unit} (clauses of length 1). If an empty clause is found, then we have derived falsity, and hence Γ was already unsatisfiable. Then, we repeatedly do the following, as long as $\Gamma_{\text{unit}} \neq \varnothing$: pick

*https://github.com/HarryBryant99, ORCID: 0009-0008-9926-8678
†https://www.swansea.ac.uk/staff/m.seisenberger/, ORCID: 0000-0002-2226-386X
‡https://csetzer.github.io/, https://www.swansea.ac.uk/staff/a.g.setzer/, ORCID: 0000-0001-5322-6060

one unit clause $\{x\} \in \Gamma_{\text{unit}}$ and remove it from Γ_{unit}. Next, we carry out unit clause resolution with all clauses c in $\Gamma_{\text{unit}} \cup \Gamma_{\text{nunit}}$. If c contains x, then it is implied by $\{x\}$ and is therefore removed. Otherwise, if c contains $\neg x$, then a unit resolution of c with $\{x\}$ derives $c' := c \backslash \{\neg x\}$. We replace c by c'; if it has length $\geqslant 2$, it will be in Γ_{nunit}, and if it has length 1, in Γ_{unit}. If it has length 0, then we have derived $\{\}$, so the RUP inference is verified, and we exit the loop. If c does not contain x or $\neg x$, it is kept. Once we have applied unit resolution with $\{x\}$ to all the clauses in $\Gamma_{\text{unit}} \cup \Gamma_{\text{nunit}}$, we repeat the process. After each step, the literal x and its negation do not occur anymore in $\Gamma_{\text{unit}} \cup \Gamma_{\text{nunit}}$, and no new literals have been created, so eventually the loop terminates because Γ_{unit} is empty. If we have not derived $\{\}$ by then, then $\{\}$ is not derivable by unit clause propagation, thus the verification of the RUP inference fails.

At each step, all formulae in $\Gamma_{\text{unit}} \cup \Gamma_{\text{nunit}}$ are derivable from Γ using unit clause resolution; therefore, they are entailed. If the procedure succeeds, then Γ' entails falsity and is therefore unsatisfiable. Thus, by classical logic (we have tertium non datur for the Boolean variables) it follows that Γ entails $\{x_1, \ldots, x_n\}$. We leave the full proof of completeness to a future article.

We have formalised the logic in Rocq [23] (formally called Coq [10]) and written a procedure that checks the proofs from Z3 and ensures that all RUP inferences are correct. For examples using more complex logical formulae or involving other data structures, such as integers supported by Z3, our checker verifies their correctness as well. However, verification of these rules is left for future work. For CNF formulae, Rocq creates only assumptions, deletions, subsumptions, and RUP inference rules. For RUP inferences, our system currently creates proofs which derive falsity from the assumptions and negated unit clauses using unit resolution. We have a proof in Rocq that if this proof is correct, the assumptions plus the negated unit clauses entail falsity and, therefore, the assumptions entail the formula in question. If all the generated proofs are correct, we have a proof in Rocq that the Z3 proof is correct. Therefore, if Z3 returns unsatisfiable, the assumptions are shown to be unsatisfiable.

Functions to check RUP inferences can be extracted from Rocq [23] into executable code using Rocq's extraction mechanism [21], typically to OCaml or Haskell. Extraction to other languages is possible, for example, C using the Codegen package [24]. Extraction to C supports basic types like numbers and lists, but complex types need extra handling or may not be supported. This is problematic for dependent types or higher-order functions lacking C equivalents.

Currently, proofs of correctness for RUP rely on generating all intermediate resolution proofs for each RUP inference. In fact, generating these proofs may be desirable when working with critical systems. Although having a proof that the checker is correct provides a high level of trust, there remains a remote possibility that an inconsistency in Rocq was used. Genuine bugs are occasionally detected in theorem provers. Therefore, having independently verifiable proof logs would allow for an even higher level of trust. The additional generated intermediate resolution proofs make it easier and therefore more trustworthy to verify the RUP proofs.

As a prototype, we are working on the verification of RUP inferences and Z3 proofs in Agda (see the GitHub repository [6]). Induction-recursion, as supported by Agda, is very beneficial in this project: we define proofs inductively while recursively deriving their conclusion. As a first step, we created a resolution proof of falsity from the assumptions and negated literals, provided that the RUP inference is correct. Therefore, these assumptions and negated literals are unsatisfiable. Although it is not of direct use for the industrial application – Agda does not allow compiling into C – the verification in Agda will, once completed, enable the integration of Z3 proofs into Agda. This advances our effort to incorporate automated theorem proving into Agda for verifying railway interlocking systems, a collaboration between Setzer and Kanso [19, 18].

References

[1] Madhusree Banerjee, Victor Cai, Sunitha Lakshmanappa, Andrew Lawrence, Markus Roggenbach, Monika Seisenberger, and Thomas Werner. A tool-chain for the verification of geographic scheme data. In Birgit Milius, Simon Collart-Dutilleul, and Thierry Lecomte, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 211–224, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-43366-5_13.

[2] Nikolaj Bjørner. Proofs for SMT, 11 October 2022. Slides, Dagstuhl, October 11 2022. URL: https://z3prover.github.io/slides/proofs.html#/.

[3] Sascha Böhme. Proof reconstruction for Z3 in Isabelle/HOL, 2009. Slides of talk given at 7th International Workshop on Satisfiability Modulo Theories (SMT '09). URL: https://wwwbroy.in.tum.de/~boehmes/proofrec-talk.pdf.

[4] Sascha Böhme. Proof reconstruction for Z3 in Isabelle/HOL. In *Workshop on Proof Exchange for Theorem Proving (PxTP)*, 2009. URL: https://www21.in.tum.de/~boehmes/proofrec.pdf.

[5] Sascha Böhme and Tjark Weber. Fast LCF-Style Proof Reconstruction for Z3. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 179–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:https://doi.org/10.1007/978-3-642-14052-5_14.

[6] Harry Bryant, Andrew Lawrence, Monika Seisenberger, and Anton Setzer. Verification of Z3 RUP Proofs in Coq-Rocq and Agda. https://github.com/HarryBryant99/Verification-of-Z3-RUP-Proofs-in-Coq-Rocq-and-Agda, 2025. Accessed: 2025-05-09.

[7] Simon Chadwick, Phillip James, Markus Roggenbach, and Thomas Werner. Formal Methods for Industrial Interlocking Verification. In *ICIRT*, 2018. doi:10.1109/ICIRT.2018.8641579.

[8] Ran Chen, Cyril Cohen, Jean-Jacques Lévy, Stephan Merz, and Laurent Théry. Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2019.13, doi:10.4230/LIPIcs.ITP.2019.13.

[9] Alessio Coltellacci, Gilles Dowek, and Stephan Merz. Reconstruction of SMT proofs with Lambdapi. In Giles Reger and Yoni Zohar, editors, *CEUR Workshop Proceedings*, volume 3725, pages 13–23, Montréal, Canada, July 2024. URL: https://inria.hal.science/hal-04861898.

[10] Rocq community. About the Rocq Prover, Accessed 10 March 2025. URL: https://rocq-prover.org/about.

[11] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient Certified RAT Verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-63046-5_14.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. URL: https://dl.acm.org/doi/abs/10.5555/1792734.1792766.

[13] Mathias Fleury and Hans-Jörg Schurr. Reconstructing veriT proofs in Isabelle/HOL. *Electronic Proceedings in Theoretical Computer Science*, 301:36–50, August 2019. doi:10.4204/eptcs.301.6.

[14] Allen van Gelder. Verifying RUP proofs of Propositional Unsatisfiability, 2008. Slides of a talk given at ISAIM'08. URL: https://users.soe.ucsc.edu/~avg/ProofChecker/Documents/proofs-isaim08-trans.pdf.

[15] Allen van Gelder. Verifying RUP Proofs of Propositional Unsatisfiability: Have Your Cake and Eat It Too, 2008. In Proceedings of 10th International Symposium on Artificial Intelligence and

Mathematics (ISAIM'08). URL: https://users.soe.ucsc.edu/~avg/ProofChecker/Documents/proofs-isaim08-long.pdf.

[16] Eugene Goldberg and Yakov Novikov. Verification of Proofs of Unsatisfiability for CNF Formulas. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 886–891, March 2003. doi:10.1109/DATE.2003.1253718.

[17] Alan G. Hamilton. *Logic for Mathematicians*. Cambridge University Press, Cambridge, rev. ed. edition, 1988.

[18] Karim Kanso. *Agda as a Platform for the Development of Verified Railway Interlocking Systems*. PhD thesis, Dept. of Computer Science, Swansea University, Swansea, UK, August 2012. Git repository available at https://github.com/kazkansouh/agda. URL: https://csetzer.github.io/articlesFromOthers/kanso/karimKansoPhDThesisAgdaAsAPlatformForVerifiedRailways.pdf.

[19] Karim Kanso and Anton Setzer. A light-weight integration of automated and interactive theorem proving. *Mathematical Structures in Computer Science*, 26(1):129–153, 2016. doi:10.1017/S0960129514000140.

[20] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3551349.3556938.

[21] Pierre Letouzey. Extraction in Coq, an Overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms: 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008 Proceedings 4*, pages 359–369, June 2008. doi:10.1007/978-3-540-69407-6_39.

[22] Marius Minea. Conjunctive Normal Form: Tseitin Transform, 2024. H250: Honors Colloquium – Introduction to Computation. URL: https://people.cs.umass.edu/~marius/class/h250/lec2.pdf.

[23] Rocq Development Team. Rocq. https://rocq-prover.org/, 2025. Accessed: 2025-05-09.

[24] Akira Tanaka. Coq to C translation with partial evaluation. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2021, pages 14–31, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3441296.3441394.

[25] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg. doi:10.1007/978-3-642-81955-1_28.

[26] Nathan Wetzler, Marijn Heule, and Warren Hunt. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *International Conference on Theory and Applications of Satisfiability Testing*, 07 2014. doi:10.1007/978-3-319-09284-3_31.

[27] Nathan David Wetzler. *Efficient, mechanically-verified validation of satisfiability solvers*. PhD thesis, Univeristy of Texas at Austin, Austin, 2015. URL: https://repositories.lib.utexas.edu/bitstream/handle/2152/30538/WETZLER-DISSERTATION-2015.pdf?sequence=1.

Formalising Monitors for Distributed Deadlock Detection

Radosław Jan Rowicki[1], Adrian Francalanza[2], and Alceste Scalas[1]

[1] Danmarks Tekniske Universitet, Kongens Lyngby, Denmark
rjro@dtu.dk alcsc@dtu.dk
[2] University of Malta, Msida, Malta
adrian.francalanza@um.edu.mt

Introduction

Modern software applications are often implemented as networks of (micro)services that communicate via message-passing. Many common protocols rely on forms of *remote procedure calls (RPC)*, where services (acting as RPC clients) perform external calls by sending requests to other services (acting as RPC servers) and then awaiting a response. For example, the *Open Telecom Platform* (OTP) and languages like Erlang and Elixir provide widely-used *behaviours* (like `gen_server` and `gen_statem`) to the development of RPC-based services [6]. Such behaviours offer what we dub *single-threaded RPC (SRPC)*, where each service handles one request at a time and remains idle while waiting for a response to a request that it sent; similar SRPC patterns can be found in popular actor frameworks like Akka/Pekko [7].

Such systems can run into *deadlocks* if a group of services end up waiting on each other in a circular dependency. Correctly identifying and fixing such deadlocks can be hard: in large distributed systems with high traffic, observers may see that part of the system is unresponsive and some requests time out, but these symptoms can be mistakenly attributed to performance issues. Deadlocks can be potentially prevented via static analysis [11, 15, 12, 13] — although this requires access to the source code of the whole system (which may not be available) and can produce false positives (which may be undesirable). In these cases, runtime verification via *monitors* [2, 8] can be a more practical method for identifying deadlocks as they occur.

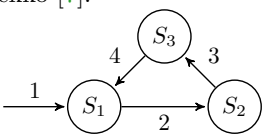


Figure 1: Deadlock example. The numbers show the order in which requests are sent.

A monitor is a process that oversees a service in order to identify faulty states. We are interested in developing *distributed* deadlock detection monitors that do not introduce centralised bottlenecks; moreover, we require our monitors to be *black-box* and *outline*, i.e., they can only observe the incoming/outgoing messages of each service without access to its internals.

In this work, we present the Coq mechanisation[1] of a theory that formalises networks of SRPC services with and without monitors, and a distributed black-box deadlock detection monitoring algorithm (inspired by [5, 14]) which we prove *sound* and *complete*. We also provide a framework (with an API inpired by Erlang and Elixir `gen_servers`) to conveniently model SRPC-based distributed applications for which we automatically construct correct deadlock monitors. To the best of our knowledge, we provide the first mechanised correctness proof of a distributed deadlock detection algorithm.

[1] https://github.com/radrow/dlstalk-coq

Mechanisation and Results

We define networks as functions from a set of *names* to states of services. In our model, communication is asynchronous, meaning that services never block when sending a message. Services are programmed directly in Gallina[10] via a coinductive DSL providing primitives for sending and receiving messages, inspired by process calculi: see fig. 2, where the `option` continuation in `Recv` allows the selective reception of messages based on senders. This design makes our model remarkably expressive while keeping our formalisation focused on communication semantics. Moreover, Coq entirely covers bindings and substitutions, which can notoriously complicate proofs if otherwise embedded [3, 1, 4]. On top of this DSL, we specify `SRPC` as a coinductive property that classifies service states as either: `Ready` while the service awaits requests, `Work`-ing when it actively processes a request, and `Lock`-ed if it awaits a response from another service.

```
CoInductive Proc := (* Service implementation *)
| Recv (select : Name → option (Val → Proc))
| Send (to : Name) (msg : Val) (P : Proc)
| Tau (P : Proc).

Record Serv := (* service *)
  { i_que : list Msg; proc : Proc; o_que : list Msg}.
Record MServ := (* monitored service *)
  { m_que : list Msg; state : MProc; serv : Serv}.
```

Figure 2: Service DSL as defined in Coq.

We implement monitoring by *instrumenting* each service with a monitor process M and a monitor message buffer $\hat{q}$. Our monitors are *intercepting*, i.e., they act as proxies between their services and the network. Consequently, all incoming and outgoing messages pass through the monitor and its buffer, allowing the monitor M to observe the service's communication, update its state, and exchange messages with other monitors. Figure 3 illustrates how a monitored service interacts with the rest of the network.

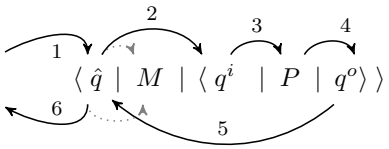


Figure 3: Journey of messages in a monitored service: (1) a message from the network reaches the monitor buffer $\hat{q}$; (2) the message is forwarded to the service's input queue q^i; (3) the service process P receives the message; (4) the service sends some message via its output queue q^o; (5) the output message reaches the monitor queue $\hat{q}$; (6) the message is forwarded to the network. The dotted arrows mean that the monitor observes messages (2) and (6), but it cannot observe the other message exchanges.

Instrumentations (`instr`) are functions that transform unmonitored SRPC networks (of type `Net`) into monitored ones (of type `MNet`) by equipping each service with a monitor process and buffer. *Vice versa*, `deinstr` "strips" monitors from a monitored network. We prove that instrumentation is *transparent*, i.e., it does not introduce nor suppress behaviours w.r.t. the original network. We formalise transparency as an *operational correspondence* [9] (proving soundness and completeness) between monitored and unmonitored networks: see Coq theorems in fig. 4a, where a path is a sequence of reduction steps.

```
Variable apply_instr (i : instr) : Net → MNet.
Coercion apply_instr : instr >-> Funclass.

Theorem transp_sound :
  ∀ (N₀ : Net) (i₀ : instr) path' (MN₁ : MNet),
    (i₀ N₀ =[ path' ]⇒ MN₁) →
  ∃ path, (N₀ =[ path ]⇒ deinstr MN₁).

Theorem transp_complete :
  ∀ (N₀ N₁ : Net) path (i₀ : instr),
    (N₀ =[ path ]⇒ N₁) →
  ∃ path' (i₁ : instr), (i₀ N₀ =[ path' ]⇒ i₁ N₁).
```

(a) Transparency of monitoring.

```
Definition detect_sound (N₀ : Net) (i₀ : instr) :=
  ∀ path' MN₁, (i₀ N₀ =[ path' ]⇒ MN₁) ∧ reports_deadlock MN₁ →
  ∃ path, (N₀ =[ path ]⇒ deinstr MN₁) ∧ has_deadlock (deinstr MN₁).

Definition detect_complete (N₀ : Net) (i₀ : instr) :=
  ∀ path N₁, (N₀ =[ path ]⇒ N₁) ∧ has_deadlock N₁ →
  ∃ path' (i₁ : instr),
    (i₀ N₀ =[ path' ]⇒ i₁ N₁) ∧ reports_deadlock (i₁ N₁).

Theorem gen_net_instr_correct : ∀ dapp,
    detect_sound (gen_instr (gen_net dapp))
  ∧ detect_complete (gen_instr (gen_net dapp)).
```

(b) Correctness of deadlock detection.

Figure 4: Key statements of the project as formalised in Coq.

Concretely, each monitor M implements a deadlock detection algorithm that, based on the incoming/outgoing messages it observes, estimates the current state of the service it oversees.

In particular, if it infers that the service is awaiting a response, it sends *probes* (in the style of [5, 14]) to the monitors of all services from which it receives a request. Monitors communicate by forwarding such probes; if a monitor receives back a non-outdated probe that it has previously emitted, then it concludes that there is a dependency cycle and reports a deadlock.

Following runtime verification standards, we define *correctness* of monitors in fig. 4b as *soundness* (every reported deadlock is real) and *completeness* (all deadlocks are eventually reported, assuming fairness of components [16]). We finally prove that any distributed application modelled with our Erlang/Elixir `gen_server`-inspired framework (`gen_net dapp`) can be correctly instrumented (`gen_instr`) and monitored; we achieve this by finding and proving invariants that characterise correct instrumentations. All proofs are successfully mechanised in Coq.

Acknowledgement

This work was partially supported by the Independent Research Fund Denmark project "Hyben."

References

[1] Beniamino Accattoli, Horace Blanc, and Claudio Sacerdoti Coen. Formalizing Functions as Processes. In *ITP 2023 - 14th International Conference on Interactive Theorem Proving*, Bialystok, Poland, July 2023. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.ITP.2023.5.

[2] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018. doi:10.1007/978-3-319-75632-5_1.

[3] Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Logical Methods in Computer Science*, Volume 5, Issue 2, June 2009. doi:10.2168/lmcs-5(2:16)2009.

[4] Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Legesse Tirore, Martin Vassor, Nobuko Yoshida, and Daniel Zackon. The concurrent calculi formalisation benchmark. In Ilaria Castellani and Francesco Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *Lecture Notes in Computer Science*, pages 149–158. Springer, 2024. doi:10.1007/978-3-031-62697-5_9.

[5] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, May 1983. doi:10.1145/357360.357365.

[6] Ericsson AB. *Erlang/OTP System Documentation 14.2.5.8*, chapter 10, pages 313–344. 2025. Accessed: 2025-03-10. URL: https://www.erlang.org/docs/26/pdf/otp-system-documentation.pdf.

[7] Erlang/OTP Team. *Apache Pekko gRPC*. Apache Software Foundation, 2025. Accessed: 2025-03-10. URL: https://pekko.apache.org/docs/pekko-grpc/current/index.html.

[8] Adrian Francalanza. A theory of monitors. *Information and Computation*, 281:104704, 2021. doi:10.1016/j.ic.2021.104704.

[9] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010. doi:10.1016/J.IC.2010.05.002.

[10] Gérard Huet. The Gallina specification language: A case study. In Rudrapatna Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 229–240, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. doi:10.1007/3-540-56287-7_108.

[11] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998. doi:10.1145/276393.278524.

[12] Naoki Kobayashi. A new type system for deadlock-free processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006. doi:10.1007/11817949_16.

[13] Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017. doi:10.1016/j.ic.2016.03.004.

[14] Don P. Mitchell and Michael J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, page 282–284, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800222.806755.

[15] Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2603088.2603116.

[16] Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019. doi:10.1145/3329125.

Fair Termination for Resource-Aware Active Objects

Francesco Dagnino[2], Paola Giannini[3], Violet Ka I Pun[1], and Ulises Torrella[1]

[1] Western Norway University of Applied Sciences
[2] University of Genoa
[3] University of Eastern Piedmont

Active object systems [13, 14, 6] are the object-oriented instantiation of the actor model [2]. They provide a useful abstraction of distributed systems with asynchronous communications, which are represented as collections of objects (actors) interacting through asynchronous method calls. This means that a method invocation corresponds to sending a message to the receiver object that will eventually handle it, by running the method body. Thus, the invocation does not block the execution of the caller, but immediately returns a future, which can be subsequently used for synchronizing with the receiver and accessing the result of the call.

Among other applications, this model provides the formal basis for workflow modelling and analysis [3, 4], where models capture the behaviour of the internal (resource-sensitive) workflows of organisations. Workflows are processes that handle business cases and are primarily demanded to be terminating, hence resolve the business case (a customer order, a service ticket, etc.). In particular, the modelling language proposed in [3] is based on the ABS language [14], an active object language with multithreaded actors and a future-based cooperative scheduling paradigm. This means that each actor can handle multiple messages at a time, by explicitly yielding control on future **await** statements.

Besides this interaction mechanism, a workflow modelling language also needs to take into account *passive/informational resources* [19]. These resources move through processes while performing a workflow, undergo transformations, can be created or destroyed and crucially have a *limited availability* according to the specification domain. The interplay of asynchronous message passing, cooperative multithreading and resource management makes it challenging to ensure that a system can complete its task. For instance, if a thread tries to access a resource that is not available, it remains stuck as it cannot yield control, thus preventing the whole system from successfully terminating.

The contributions of this work are the development of a core calculus for resource-aware active objects together with a type system ensuring that well-typed programs are *fairly terminating* [9], that is, all their fair executions terminate, under a suitable fairness assumption. To achieve this, we combine techniques from graded semantics and type systems [7, 1, 8, 15, 5, 18], which are quite well understood for sequential programs, with those for fair termination [10, 11, 9, 12] , which have been developed for synchronous sessions.

More in detail, we model resources as *graded constants* r^g where r is a name identifying the resource and g is a grade describing the availability of the resource and thus constraining its usage. For instance, a resource can be used a fixed number of times or in a private or public mode. In our calculus, each actor owns a resource environment ρ, containing graded resources which can be accessed by threads of that actor. The language provides constructs through which a thread of an actor can hold and release resources from its resource environment. Notably, the reduction of a **hold** statement asking for r^g, i.e., "g copies" of the resource r, is stuck if the amount, i.e., the grade, of the resource r in the environment ρ of the actor is not enough to produce r^g. Finally, it is important to note that the introduction of graded resources has an impact also on the synchronization mechanism. Indeed, typically futures can be accessed an arbitrary number of times [17, 16, 14]. However, this is not the case in our setting because futures may contain graded resources and so, by copying the future, we would copy its content

as well, leading to a violation of the constraint on the resource usage expressed by the grade. To overcome this issue, we treat futures *linearly*, allowing for them to be read only once.

In order to ensure the correct use of resources, we endow our calculus with a graded type system [7, 1, 8, 15, 5, 18]. Besides basic types, we have graded types for resources and future types. Then, the typing judgment for expressions has the following shape: $\Phi;\ \Sigma;\ \Gamma \vdash e : T; \Phi'$, where the resource context Φ tracks the resource requirements the expression poses on each actor and it is handled like a graded context, the future context Σ tracks, in a linear way, the futures that the expression will read, and an almost standard variable context Γ where each variable is handled in a linear, graded or unrestricted way depending on its type. An expression is assigned a type T and a "release context" Φ' with the resources that the expression will release into the system. This judgment allows us to "chain" resource production and consumption in the sequential composition, enabling a form of reuse of resources, and also expose this information on the method type. Thus, given that the body of a method is an expression, its return type will be T, corresponding to the return value of the body, plus the resources released into the system Φ'. Then, on account of all methods being asynchronous, the type of a method call will be a Future $\mathbf{Fut}\langle T, \Phi'\rangle$. Hence, an **await** expression on a future of such type will have type T and release context Φ', so that the resources released by the method call can be reused by the process accessing its result.

Futures are the lone communication mechanism of processes. To avoid circular dependencies on futures we enforce a left to right future dependency on configurations by typing processes with judgments of shape: $\Phi;\ \Sigma \vdash P :: \Sigma'$, where there is a left-hand side resource context Φ for the required resources of the process, a left-hand side future context Σ for used futures, and a right-hand side future context Σ' for the produced futures. Processes are composed by parallel composition, whose typing rule sums up the resource contexts of the two parallel processes and checks that the process on the right only reads futures produced by the process on the left, thus ensuring that the dependency graph on futures is acyclic.

The main result of this work is a proof that well-typed configurations are fairly terminating. This is achieved applying a proof technique from [9] ensuring that to obtain fair termination it is enough to prove the standard subject reduction and weak termination of well-typed configurations. The latter means that every well-typed configuration admits a terminating execution. Note that, since our calculus supports a non-deterministic choice operator, this property does not forbid non-termination, but it ensures that termination is always possible. To prove this result, following [9], we annotate typing judgments with a measure, taken from a well-founded poset, and prove that there is always a reduction step making such measure decrease. The proof of subject reduction poses some challenges as well. Indeed, it does not hold for standard graded semantics and type systems [8, 5] because resource consumption in graded semantics usually is non-deterministic, hence only a form of "may subject reduction" can be proved, where well-typedness after a step is not guaranteed. Therefore, in order to recover subject reduction, we need to define a semantics where resources are consumed in a deterministic way and this requires an extension of the usual algebraic structure of grades adopted in the literature. Finally, by fair termination we guarantee that every well-typed system can always successfully terminate and so it is resource safe and never stuck.

This work is a first step towards integrating resource-awareness by grading in active objects systems, ensuring strong behavioural properties like fair termination. Notably, we would like to relax the linearity constraint on futures by treating them in a graded way as well. Moreover, it would be interesting to investigate a system where threads can yield control also on **hold** statements, so that they could be paused until the required resources are available, thus reducing the possibility for an actor of being stuck.

References

[1] Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proc. ACM Program. Lang.*, 4(ICFP):90:1–90:28, 2020.

[2] Gul A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems (Parallel Processing, Semantics, Open, Programming Languages, Artificial Intelligence)*. PhD thesis, University of Michigan, USA, 1985.

[3] Muhammad Rizwan Ali, Yngve Lamo, and Violet Ka I Pun. Cost analysis for a resource sensitive workflow modelling language. *Sci. Comput. Program.*, 225:102896, 2023.

[4] Muhammad Rizwan Ali, Violet Ka I Pun, and Guillermo Román-Díez. Easyrpl: A web-based tool for modelling and analysis of cross-organisational workflows. *CoRR*, abs/2502.20972, 2025.

[5] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. Resource-aware soundness for big-step semantics. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1281–1309, 2023.

[6] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In Marco Bernardo and Einar Broch Johnsen, editors, *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*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.

[7] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *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*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014.

[8] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021.

[9] Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. *J. Log. Algebraic Methods Program.*, 139:100964, 2024.

[10] Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022.

[11] Luca Ciccone and Luca Padovani. An infinitary proof theory of linear logic ensuring fair termination in the linear π-calculus. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[12] Francesco Dagnino and Luca Padovani. small caps: An infinitary linear logic for a calculus of pure sessions. In Alessandro Bruni, Alberto Momigliano, Matteo Pradella, Matteo Rossi, and James Cheney, editors, *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, Milano, Italy, September 9-11, 2024*, pages 4:1–4:13. ACM, 2024.

[13] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017.

[14] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*,

volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.

[15] Ugo Dal Lago and Francesco Gavazzo. A relational theory of effects and coeffects. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022.

[16] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

[17] Siva Somayyajula and Frank Pfenning. Type-based termination for futures. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, volume 228 of *LIPIcs*, pages 12:1–12:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[18] Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. Effects and coeffects in call-by-push-value. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1108–1134, 2024.

[19] Michael zur Muehlen. Organizational management in workflow applications - issues and perspectives. *Inf. Technol. Manag.*, 5(3-4):271–291, 2004.

Mechanized safety of Jolteon consensus in Agda

Orestis Melkonian, Mauro Jaskelioff, and James Chapman

Input Output, Global (IOG)

Introduction. Consensus protocols for distributed systems ensure that all participants agree on some state, which is often realised as a common order of blocks on a chain. They are at the core of blockchain technology, where any mistake in the protocol design or implementation could result in huge economic losses. Therefore, it is of utmost importance to provide strong guarantees of its correctness.

At the same time, the area of consensus protocols is rapidly moving, as evidenced by the amount of new and improved protocols that have appeared in the last few years [4, 9, 6, 3, 1, 5, to cite a few]. Hence, it might not be possible to aim both for a complete formally proven implementation and stay at the forefront of technological development. A good compromise is to formally verify the design of the protocol, and use this formalization as an oracle for testing. The formalization becomes the ground truth, and all implementations should follow it. However, this approach entails an additional requirement: the formalization should be readable by engineers which might not be well versed in the formalization language or its abstractions.

We present a formalization of the Jolteon consensus protocol [6], a modern consensus protocol of the BFT (Byzantine Fault Tolerance) family [2], and we mechanize its proof of safety. Safety for a consensus protocol means that consistency is always maintained (*i.e.* there are no diverging chains). The formalization is written with readability in mind, and aims to stay close to the paper description. All of our results are mechanized in the Agda proof assistant [7].

A formal definition of the Jolteon protocol. We present a readable Agda specification of the Jolteon consensus protocol [6], with the aim of mechanizing its pen-and-paper **proofs** of important properties such as safety, as well as having a rigid ground truth against which we can **test** actual implementations.

First, we assume the usual cryptographic primitives (hash functions, signatures) and the BFT-specific setup of a fixed number `n` of replicas/participants which contains an *honest majority* [2]. Our further formal development is parameterised over these assumptions.

Each participant, honest or not, has an identifier `Pid = Fin n`.

The description of the protocol takes the form of a binary step relation that formally expresses valid transitions between states of the **global** system. This global level mostly deals with generic scaffolding that every BFT consensus protocol would have to provide, such as the message-passing `Deliver` rule that enables communication between participants, the `WaitUntil` rule that advances time (as long as it does not break the assumption of the underlying network model of message delays having an upper bound Δ), and the `DishonestStep` rule which gives dishonest participants freedom to send any message as long as it doesn't involve forging signatures.

```
data _—→_ (s : GlobalState) : GlobalState → Type where
```

```
Deliver : ∀ {tm} →
  tm ∈ s .networkBuffer
  ──────────────────────
  s —→ deliverMsg s tm

WaitUntil : ∀ t →
  • All (λ (t′ , _) → t ≤ t′ + Δ) (s .networkBuffer)
  • currentTime s < t
  ──────────────────────────────
  s —→ record s { currentTime = t }
```

```
DishonestStep : ∀ m →
  • NoSignatureForging m s
  ──────────────────────
  s —→ broadcast m s
```

...

Apart from handling the message-passing aspect of the network, the global state also keeps track of each honest replica's **local** state, which contains all protocol-specific information such as the current round, the most recent certificate the replica has seen, etc. The behaviour of honest replicas is modelled with another step relation where the specific rules of the protocol manifest, such as how each epoch's honest leader *proposes* a block, or under which conditions a replica considers a chain to be *final*:

```
data _⦂_⊢_—_—→_ (p : Pid) (t : Time) (ls : LocalState) : Maybe Message → LocalState → Type where
  ProposeBlock : ∀ txs →
    let L = roundLeader (ls .r-cur)
        b = mkBlockForState ls txs
        m = Propose (sign L b)
    in
    • p ≡ L
    ──────────────────────────
    p ⦂ t ⊢ ls — just m —→ ls
```

```
  Commit : ∀ b b′ ch →
    • b  -certified-∈- ls .db
    • b′ -certified-∈- ls .db
    • (b′ :: b :: ch) •∈ ls .db
    • length ch > length (ls .final)
    • b′ .round ≡ 1 + b .round
    ──────────────────────────────────────────
    p ⦂ t ⊢ ls — nothing —→ record ls {final = b :: ch}
```

...

Proving safety. Taking the reflexive-transitive closure of the global step relation —→ above leads to a notion of execution trace for such a protocol. That way, we can prove state invariants that hold for every *reachable* state in those traces, that is all states for which there is a valid sequence of steps from the initial state.

Once we have proven several state invariants, *e.g.* expressing a certain connection between different pieces of the state, we were able to faithfully transcribe the original paper proof of `safety` (otherwise known as *consistency*) in Agda. Safety states that two honest participants never have diverging chains that they both consider final:

```
safety : ∀ s b p b′ p′ →
  • Reachable s
  • b  ∈ (s @ p)  .final
  • b′ ∈ (s @ p′) .final
  ──────────────────────
  (b ←—* b′) ⊎ (b′ ←—* b)
```

Testing. We further prove several decidability results in Agda about all the logical constructions we used thus far to define the protocol, such as the chain finalization conditions.

The purpose of these proofs is two-fold. First, they allow us to construct and type-check example traces without manually providing proofs for each rule hypothesis, since our model is *computable* and therefore one can leverage *proof-by-computation* [8] to automate proofs on closed examples that contain no variables.

Moreover, once extracted/compiled to a target general-purpose language, these proofs of decidability become *decision procedures* that can be utilised by a **conformance testing** pipeline that aims to ensure that a given replica's protocol implementation conforms to the formal specification.

Future work. We plan to proceed with also proving **liveness**, a crucial property to ensure the protocol makes progress in a timely fashion. Apart from having to model some additional aspects that relate to the notion of *time*, we expect the methodology we previously employed for safety to also extend adequately to liveness.

Finally, we plan to investigate different testing approaches (*e.g.* testing a local honest replica against the local step relation *versus* testing the whole testing environment against the global step relation), as well as variants of the protocol that have more optimal characteristics which we would prove "equivalent" to the original protocol (*i.e.* the original properties of safety and liveness still hold in the optimized protocol).

References

[1] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. 10 2017. doi:10.48550/arXiv.1710.09437.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

[3] Benjamin Y. Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. In Guy N. Rothblum and Hoeteck Wee, editors, *Theory of Cryptography - 21st International Conference, TCC 2023, Taipei, Taiwan, November 29 - December 2, 2023, Proceedings, Part IV*, volume 14372 of *Lecture Notes in Computer Science*, pages 452–479. Springer, 2023. doi:10.1007/978-3-031-48624-1_17.

[4] T-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 2018:981, 2018. URL: https://api.semanticscholar.org/CorpusID:53238268.

[5] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader bft via optimistic proposals, 2024. URL: https://arxiv.org/abs/2401.01791, arXiv:2401.01791.

[6] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In Ittay Eyal and Juan A. Garay, editors, *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022. doi:10.1007/978-3-031-18283-9_14.

[7] Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[8] Paul Van Der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, pages 157–173. Springer, 2012.

[9] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331591.

Canonical Bidirectional Typing via Polarised System L

Zanzi Mihejevs

Glasgow Lab for AI Verification

1 Abstract

What is the relationship between polarity and bidirectional typing? It has long been observed that there is a connection between the two[Kri], but the precise relationship has remained unclear. Moreover, it has been argued [McB] that the link itself is a red herring, and that bidirectional typing is better explained not by polarity but by chirality - the duality between producers and consumers.

Polarised System L [Dow17] is a type theory that combines both dualities - the positive fragment is driven by a cut between a primitive producer and a pattern, and the negative fragment is driven by a cut between a primitive consumer and a co-pattern.

Remarkably, linear System L admits a canonical bidirectional typing discipline based on a combination of ideas from both standard and co-contextual typing, giving us a "bi-contextual" typing algorithm.

This lets us equip a type system based on classical linear logic - containing all four connectives and derivable implication and co-implication - with a bidirectional discipline where all typing annotations are exclusively limited to shifts between sythesisable and checkable expressions.

2 Introduction

Bidirectional typing has established itself as an effective approach to developing type-driven type-checking algorithms[DK21], however equipping a type system with a bidirectional discipline can still be more of an art than a science. This is especially true when it comes to type systems with non-trivial elimination forms, such as those involving binding. Prior work [MRK22] on polarised approaches to bidirectional typing has shown that utilising polarity in algorithmic type-checking can significantly simplify the type inference algorithm. We show how using System L rather than CBPV allows us to take this even further and give a decidable type-inference algorithm for a type-system based on full linear logic.

The advantage of System L over CBPV is that while both calculi have a notion of polarity, CBPV's split between values and computations is tied to the polarity of each type - value judgements correspond to positive types, and computation judgements correspond to negative types. On the other hand, in polarised System L both positive and negative types each have a pair of judgements corresponding to their own notion of producer and consumer terms. Moreover, each polarity has its own notion of focused judgement, which we call its 'principal chirality' - positive types are focused on producer terms, while negative types are focused on consumers.

3 Principal Chirality

The key idea of our approach is that each polarity has a 'principal chirality' corresponding to the focused judgement, and the 'auxiliary chirality' corresponding to the unfocused judgement.

Type-checking each polarity proceeds by first synthesizing the type of the judgement corresponding to the principal chirality, then checking the type of the judgement of the auxiliary chirality against the synthesized type. The crucial difference between the two phases is in the way that information gets propagated in the principal and auxiliary contexts. The principal context is built-up bottom-up using the co-contextual [EBK+15] approach (which means that positive variables and negative co-variables are both checkable), while the auxiliary contexts are built-up top-down (so negative variables and positive co-variables are synthesised). This combination of contextual and co-contextual typing is why we call this a 'bi-contextual' typing discipline.

4 Typing Algorithm

Remarkably, once we adopt the bi-contextual typing discipline, we discover that System L requires almost no further modifications to make it fit into this framework. The checkable/synthesisable discipline of each connective is fully determined by its canonical place in polaraised System L, and the only modification that we need to do is to add annotations on two shifts.

To type-check the positive fragment, we start with a cut between a producer and a pattern.

$$\frac{\Gamma \vdash \mathit{producer} \Leftarrow A | \Delta \quad \Gamma' | \mathit{pattern} \Rightarrow A \vdash \Delta'}{<\mathit{producer}|\mathit{pattern}> : (\Gamma, \Gamma' \vdash \Delta, \Delta')}(\text{Cut})$$

We first synthesise the type of the pattern, then check the type of the producer against the synthesised type, after which we are done. This covers all rules in the positive fragment.

Dually, to type-check the negative fragment, we start with a cut between a consumer and a corresponding co-pattern.

$$\frac{\Gamma, \mathit{copattern} \Rightarrow A \vdash \Delta \quad \Gamma' \vdash \mathit{consumer} \Leftarrow A, \Delta'}{<\mathit{copattern}|\mathit{consumer}> : (\Gamma, \Gamma' \vdash \Delta, \Delta')}(\text{Cut})$$

We first synthesise the type of a co-pattern, then check the type of the producer against the synthesised type, after which we are done. This covers the entirety of the negative fragment.

Finally, in order to type the full calculus, we need to say what happens to negation and the shifts. Curiously, negation inverts the polarity of each connective, but it does not change their principal chirality - so a value becomes a covalue and vice-versa, and patterns swap with copatterns. This requires no additional annotational burden as it means that checkable judgements remain checkable, and synthesisable judgements remain synthesisable.

The most interesting thing happens when we consider the shifts. The shifts amount to swapping *both* the polarities and principal chiralities of a judgement. On one side, a copattern embeds into a value, and a pattern embeds into a covalue. This corresponds to embedding a synthesisable judgement into a checkable one, and can be done with no issue. But the other way around - embedding a (co)value into a (co)pattern requires us to synthesise the type of a checkable judgement, which is precisely the only two places where we need to place a typing annotation.

References

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys (CSUR)*, 54(5):1–38, 2021.

[Dow17] Paul Downen. *Sequent Calculus: A Logic and a Language for Computation and Duality.* PhD thesis, University of Oregon, 2017.

[EBK+15] Sebastian Erdweg, Oliver Bračevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 880–897, 2015.

[Kri] Neel Krishnaswami. Polarity and bidirectional typechecking. Available at ̈https://semantic-domain.blogspot.com/2018/08/polarity-and-bidirectional-typechecking.html".

[McB] Conor McBride. Basics of bidirectionalism. Available at ̈https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/".

[MRK22] Henry Mercer, Cameron Ramsay, and Neel Krishnaswami. Implicit polarized f: local type inference for impredicativity. *arXiv preprint arXiv:2203.01835*, 2022.

An existential-free theory of arithmetic in all finite types

Makoto Fujiwara

Tokyo University of Science, Japan
makotofujiwara@rs.tus.ac.jp

Abstract

We introduce an existential-free theory of arithmetic in all finite types. The theory is sufficient for the soundness of the modified realizability interpretation, and hence, it can be regarded as a constructive foundational theory.

The study of arithmetic in all finite types emerged as a response to foundational problems in so-called Hilbert's program, which was an attempt to show the consistency of (formalized) mathematical theories using only "finitistic" reasoning about finite mathematical objects. By the achievement of Gödel's second incompleteness theorem, however, it was found that any consistent recursively enumerable theory which contains elementary arithmetic, does not prove its own consistency. Since finitistic reasoning (no matter what it may be) about finite mathematical objects can be formalized in elementary arithmetic, Gödel's second incompleteness theorem suggests that Hilbert's program is unattainable. Then Hilbert's program was modified to show the consistency of arithmetical theories in a manner which is as finitistic as possible. In particular, Gödel himself worked on this modified Hilbert's program. In [2] (while it is known that Gödel had the idea already in late 1930's), he introduced a quantifier-free theory of arithmetic in all finite types and reduced the consistency of first-order arithmetic PA to the consistency of a quantifier-free theory, which is known as Gödel's T. Gödel's T is a natural extension of primitive recursive arithmetic PRA which is regarded as a finitistic theory for functions over natural numbers, to functionals in all finite types, and is (more or less) acceptable as a theory which reflects finitistic standpoint in an extended sense.

On the other hand, another controversial standpoint in foundations of mathematics is constructivism. Constructivism asserts that it is necessary to construct a witness (a mathematical object) in order to prove that something exists. In the viewpoint of constructivism, a mathematical object does exist only when one can give a construction of the object. On the other hand, in the standard finitism, a mathematical object does not exist unless it can be constructed from natural numbers in a finite number of steps. Then finitism can be seen as an extreme form of constructivism. In [2], Gödel devotes a lot of space to argue that his ground of his consistency proof of arithmetic (namely, the consistency of the above mentioned T) is more plausible than Heyting's justification of his arithmetic HA, which is a counterpart of PA based on intuitionistic logic.

From a modern perspective, what Gödel established in his papers [1, 2] can be regarded as follows: In [1], he gave a translation, which is called the "negative translation" nowadays, of formal proofs of PA into formal proofs of HA. In [2], he gave a translation, which is called the "Dialectica (or functional) interpretation" nowadays, of formal proofs of HA into formal proofs of his quantifier-free theory T in all finite types. By these two steps, the consistency of PA can be reduced to the consistency of T finitistically. If one considers classical and intuitionistic arithmetic in all finite types, Gödel's achievements show the following:

1. The consistency of classical arithmetic in all finite types PA^ω is finitistically reducible to the consistency of intuitionistic arithmetic in all finite types HA^ω.

2. The consistency of HA^ω is finitistically reducible to the consistency of T.

On the other hand, in connection with constructivism, many kinds of realizability interpretation have been studied and investigated for extracting programs from proofs. In particular, Kreisel's modified (or generalized) realizability interpretation (cf. [4, 5]) is a sort of direct formalization of the notion of constructive proofs in the language of arithmetic in all finite types. In this work, we introduce an extensional theory of arithmetic in all finite types $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$, whose language is $\exists$-free (also called negative), namely, the language does not contain disjunction and existential quantifiers (of any finite types). Our theory $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$ is similar to Kreisel's verification theory $\mathsf{HA}^\omega_{\mathsf{NF}}$ in [5] for the soundness of the modified realizability interpretation. However, our theory $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$ is consistent with classical logic, while $\mathsf{HA}^\omega_{\mathsf{NF}}$ contains some axiom scheme on continuity which is inconsistent with classical logic, In fact, $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$ is a subtheory of the extensional variant $\mathsf{E\text{-}HA}^\omega$ (see [3, Chapter 3]) of HA^ω, and T can be seen as a subtheory of $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$. In this sense, $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$ is a theory in between $\mathsf{E\text{-}HA}^\omega$ and T. Now let $\mathsf{E\text{-}PA}^\omega$ be a classical extension of $\mathsf{E\text{-}HA}^\omega$. With respect to the Gödel-Gentzen negative translation [7, Section 1.10.2] and the modified realizability interpretation [3, Chapter 5], by induction on the length of given derivations, one can show the following:

Theorem 1. *Let A be an arbitrary $\mathsf{E\text{-}PA}^\omega$-formula and Δ be an arbitrary set of $\mathsf{E\text{-}PA}^\omega$-formulas. If $\mathsf{E\text{-}PA}^\omega + \Delta$ proves A, then $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}} + \Delta^N$ proves the Gödel-Gentzen negative translation A^N of A, where Δ^N is the set of the negative translations of the formulas in Δ.*

Theorem 2 (Soundness of the modified realizability interpretation)**.** *Let A be an arbitrary $\mathsf{E\text{-}HA}^\omega$-formula and Δ_{ef} be an arbitrary set of $\exists$-free formulas of $\mathsf{E\text{-}HA}^\omega$. If $\mathsf{E\text{-}HA}^\omega + \mathrm{AC}^\omega + \mathrm{IP}^\omega_{\mathrm{ef}} + \Delta_{\mathrm{ef}}$ proves A, then one can extract a tuple of terms $\underline{t}$ of $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$ such that $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}} + \Delta_{\mathrm{ef}}$ proves $\underline{t}$ is a modified-realizer of A and all the variables in $\underline{t}$ are contained in the free variables of A, where AC^ω is the scheme of choice in all finite types and $\mathrm{IP}^\omega_{\mathrm{ef}}$ is the independence-of-premise-schema for $\exists$-free formulas in all finite types.*

That is, the negative translation and the modified realizability interpretation (finitistically) reduce the consistency of the theories to the consistency of our theory $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$. In addition, both of the negative translation and the modified realizability interpretation for $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$-formulas (namely, $\exists$-free formulas) do not change the formulas in question anymore. Because of the facts that (i) the modified realizability interpretation is a sort of direct formalization of the notion of constructive proofs, (ii) $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$ is sufficient for the verification of the soundness of the modified realizability interpretation, and (iii) the modified realizability interpretation of each theorem of $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$ is the theorem itself, our existential-free theory $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$ can be regarded as a constructive foundational theory in comparison with that T is a finitistic theory (in an extended sense). From this perspective, one may argue that Gödel firstly in [1] showed the consistency of PA based on a constructive foundation, and secondly in [2], showed that based on a finitistic foundation (in an extended sense).

In addition, we introduce an axiom scheme $\text{N-}\mathrm{AC}^\omega_{\mathrm{ef}}$, which consists of the negative translations of the instances of AC^ω in the language of $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}}$. Then the soundness of the modified realizability interpretation for the negative translation of $\mathsf{E\text{-}PA}^\omega + \mathrm{AC}^\omega$ can be verified in $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}} + \text{N-}\mathrm{AC}^\omega_{\mathrm{ef}}$, In particular, for the interpretation of the countable choice scheme $\mathrm{AC}^{\mathbb{N}}$, only the countable fragment $\text{N-}\mathrm{AC}^{\mathbb{N}}_{\mathrm{ef}}$ of $\text{N-}\mathrm{AC}^\omega_{\mathrm{ef}}$ is enough, and hence, $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}} + \text{N-}\mathrm{AC}^{\mathbb{N}}_{\mathrm{ef}}$ is a constructive counterpart of T augmented with the bar recursion in the extended finitism with respect to Spector's consistency proof of classical analysis (cf. [6]). Furthermore, we show that the modified realizability interpretation of the monotone bar induction of type $\mathbb{N}$ (Kleene's formalization of Brouwer's bar theorem) can be realized in $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}} + \text{N-}\mathrm{AC}^{\mathbb{N}}_{\mathrm{ef}}$ augmented with the bar recursion for type-$\mathbb{N}$ objects. The latter is consistent with $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}} + \text{N-}\mathrm{AC}^{\mathbb{N}}_{\mathrm{ef}}$ (relative to the extended finitism with respect to Spector) but not so with $\mathsf{E\text{-}HA}^\omega_{\mathsf{ef}} + \text{N-}\mathrm{AC}^\omega_{\mathrm{ef}}$.

References

[1] Kurt Gödel. Zur intuitionistischen Arithmetik und Zahlentheorie. *Ergebnisse eines mathematischen Kolloquiums*, 4:34–38, 1933.

[2] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[3] Ulrich Kohlenbach. *Applied proof theory: proof interpretations and their use in mathematics*. Springer Monographs in Mathematics. Springer-Verlag, Berlin, 2008.

[4] Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In *Constructivity in mathematics: Proceedings of the colloquium held at Amsterdam, 1957 (edited by A. Heyting)*, Studies in Logic and the Foundations of Mathematics, pages 101–128. North-Holland Publishing Co., Amsterdam, 1959.

[5] Georg Kreisel. On weak completeness of intuitionistic predicate logic. *J. Symbolic Logic*, 27:139–158, 1962.

[6] Clifford Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles formulated in current intuitionistic mathematics. In F. D. E. Dekker, editor, *Recursive Function Theory: Proceedings of Symposia in Pure Mathematics, volume 5*, pages 1–27. American Mathematical Society, Providence, Rhode Island, 1962.

[7] Anne S. Troelstra, editor. *Metamathematical investigation of intuitionistic arithmetic and analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, New York, 1973.

A Generalized Logical Framework

András Kovács[1] and Christian Sattler[2]

Chalmers University of Technology & University of Gothenburg, Sweden
[1] andrask@chalmers.se [2] sattler@chalmers.se

Logical frameworks (LFs [3]) and the closely related two-level type theories (2LTTs [1]) let us work in a mixed syntax of a metatheory and a chosen object theory. Here, we have a second-order view on the object theory, where contexts, variables and substitutions are implicit, and binders are represented as meta-level functions. There are some well-known limitations to LFs. First, we have to pick a model of the object theory externally. Second, since we only have a second-order view on that model, many constructions cannot be expressed; for example, the induction principle for the syntax of an object theory requires a notion of first-order model, where contexts and substitutions are explicit. Various ways have been described to make logical frameworks more expressive by extending them with modalities (e.g. [9, 4, 8, 7]). In the current work we describe an LF with the following features:

- We can work with multiple models of multiple object theories at the same time. By "theory" we mean a second-order generalized algebraic theory (SOGAT [10, 6]); this includes all type theories and programming languages that only use structural binders.
- We have both an "external" first-order view and an "internal" second-order view on each model, and we can freely switch between perspectives. All models of object theories are defined internally in the LF.
- The LF is fully structural as a type theory; no substructural modalities are used.

The Generalized Logical Framework (GLF). The basic structure is as follows.

- We have a universe U closed under the type formers of extensional type theory.
- We have $\mathsf{Base} : \mathsf{U}$, $1 : \mathsf{Base}$ and $\mathsf{PSh} : \mathsf{Base} \to \mathsf{U}$, such that each PSh_i is a universe that supports ETT. We have cumulativity: $\mathsf{PSh}_i \subseteq \mathsf{U}$. We can only eliminate from PSh_i to PSh_i. *Semantically, each* PSh_i *is a universe of presheaves over some base category represented by* i*. The terminal category is* 1.
- For convenience, we assume type-in-type everywhere, so that $\mathsf{U} : \mathsf{U}$ and $\mathsf{PSh}_i : \mathsf{PSh}_i$.
- We define $\mathsf{Cat}_i : \mathsf{PSh}_i$ as the type of categories internally to PSh_i, where types of objects and morphisms are in PSh_i. We have $\mathsf{In} : \{i : \mathsf{Base}\} \to \mathsf{Cat}_i \to \mathsf{U}$ and $\mathsf{base} : \mathsf{In}\, C \to \mathsf{Base}$. *Informally,* $\mathsf{In}\, C$ *is a type of "permission tokens" for working in a presheaf universe.*

Let us look at a basic scenario in GLF. In the empty context, we have PSh_1 as a universe of sets. Internally to PSh_1, we can define some $C : \mathsf{Cat}_1$. Now, under the assumption of $i : \mathsf{In}\, C$, we can form $\mathsf{PSh}_{(\mathsf{base}\, i)}$ as the universe of presheaves over C. Then, we may define another category D inside $\mathsf{PSh}_{(\mathsf{base}\, i)}$, and get a new universe PSh_j under the assumption of $j : \mathsf{In}\, D$. Hence, GLF at its heart is a type theory for *iterated internal categories and presheaves.*

At this point there is no interesting interaction between presheaf universes, so we proceed to specify some. Recall that the standard semantics of 2LTTs is in presheaves over a chosen model of an object theory, where a model consists of a category of contexts plus extra structure. In GLF, if we define a model M of a theory in some PSh_i universe, we would like to have a "view"

on that model internally to presheaves over M, and also a way to move between the internal view and the external M. Hence we specify that for any first-order model M of a second-order generalized algebraic theory T, we have

1. A second-order model of T internally to PSh_i, assuming $i : \mathsf{In}\, M$[1]. We write this second-order model as S_i.
2. *Yoneda embedding* as a certain family of maps from M to S_j.

We look at the example where T is pure lambda calculus. A second-order model of pure LC in PSh_i is simply $\mathsf{Tm} : \mathsf{PSh}_i$ together with an isomorphism $\mathsf{Tm} \simeq (\mathsf{Tm} \to \mathsf{Tm})$ for abstraction and application. We write lam for the former and –$– for the latter. A first-order model is a unityped category with families [2], where we write $\mathsf{Con} : \mathsf{PSh}_i$ for the type of contexts, $\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{PSh}_i$ for substitutions, $\mathsf{Tm} : \mathsf{Con} \to \mathsf{PSh}_i$ for terms, $\Gamma\, \triangleright : \mathsf{Con}$ for the extension of $\Gamma : \mathsf{Con}$ with a binding, and we have a natural isomorphism $\mathsf{Tm}\, \Gamma \simeq \mathsf{Tm}\, (\Gamma\, \triangleright)$ to represent abstraction and application. Now, assuming a first-order model M in PSh_i and $j : \mathsf{PSh}_{(\mathsf{base}\, i)}$, we can use the second-order view when working inside PSh_j. Let us define the Y-combinator as an example:

$$\begin{aligned}&\mathsf{YC} : \mathsf{Tm}_{\mathsf{S}_j}\\ &\mathsf{YC} := \mathsf{lam}_{\mathsf{S}_j}(\lambda\, f.\, (\mathsf{lam}_{\mathsf{S}_j}(\lambda x.\, x\, \$_{\mathsf{S}_j}\, x))\, \$_{\mathsf{S}_j}\, (\mathsf{lam}_{\mathsf{S}_j}(\lambda x.\, f\, \$_{\mathsf{S}_j}\, (x\, \$_{\mathsf{S}_j}\, x))))\end{aligned}$$

With a reasonable amount of sugar, we may write $\mathsf{YC} := \mathsf{lam}\, f.\, (\mathsf{lam}\, x.\, x\, x)\, (\mathsf{lam}\, x.\, f\, (x\, x))$. In other words, PSh_j now is effectively a presentation of a two-level type theory over pure LC where S_j constitutes the inner level and the ETT type formers in PSh_j constitute the outer level. Since we specify S for every second-order algebraic theory, all 2LTTs are syntactic fragments of GLF. Next, Yoneda embedding for pure LC is as follows:

$$\begin{aligned}&\mathsf{Y} : \mathsf{Con}_M &&\to ((j : \mathsf{In}_M) \to \mathsf{PSh}_j)\\ &\mathsf{Y} : \mathsf{Sub}_M\, \Gamma\, \Delta &&\simeq ((j : \mathsf{In}_M) \to \mathsf{Y}\, \Gamma\, j \to \mathsf{Y}\, \Delta\, j)\\ &\mathsf{Y} : \mathsf{Tm}_M\, \Gamma &&\simeq ((j : \mathsf{In}_M) \to \mathsf{Y}\, \Gamma\, j \to \mathsf{Tm}_{\mathsf{S}_j})\end{aligned}$$

such that Y preserves empty context and context extension, so $\mathsf{Y} \bullet j \simeq \top$ and $\mathsf{Y}\, (\Gamma\, \triangleright)\, j \simeq \mathsf{Y}\, \Gamma\, j \times \mathsf{Tm}_{\mathsf{S}_j}$, and Y preserves all other structure strictly. *Notation:* we write Λ for inverses of Y. Now, Y and Λ allow ad-hoc switching between perspectives. Let's redefine some operations in M:

$$\begin{aligned}&\mathsf{id} : \mathsf{Sub}_M\, \Gamma\, \Gamma && \mathsf{comp} : \mathsf{Sub}_M\, \Delta\, \Theta \to \mathsf{Sub}_M\, \Gamma\, \Delta \to \mathsf{Sub}_M\, \Gamma\, \Theta\\ &\mathsf{id} := \Lambda\, (\lambda\, j\, \gamma.\, \gamma) && \mathsf{comp}\, \sigma\, \delta := \Lambda\, (\lambda\, j\, \gamma.\, \mathsf{Y}\, \sigma\, (\mathsf{Y}\, \delta\, \gamma\, j)\, j)\end{aligned}$$

With reasonable amount of sugar, this might look like

$$\mathsf{id} := \Lambda\, \gamma.\, \gamma \qquad \mathsf{comp}\, \sigma\, \delta := \Lambda\, \gamma.\, \mathsf{Y}\, \sigma\, (\mathsf{Y}\, \delta\, \gamma)$$

Or, making Y implicit, we may even write $\mathsf{comp}\, \sigma\, \delta := \Lambda\, \gamma.\, \sigma\, (\delta\, \gamma)$. We can develop this into a "second-order notation" for object theories, which is nicely readable and can be rigorously elaborated into annotated GLF operations. We only give here a glimpse of what this notation could look like. The example below comes from a model construction involving models of MLTT

[1] Here we implicitly cast M to its underlying category.

as CwFs, which looks fairly obtuse with explicit substitutions and De Bruijn indices [5, Section 5]:

$$
\begin{aligned}
&\mathsf{Con}^\circ\,\Gamma && := \mathsf{Ty}\,(F\,\Gamma)\\
&\mathsf{Ty}^\circ\,\Gamma^\circ\,A && := \mathsf{Ty}\,(F\,\Gamma \triangleright \Gamma^\circ \triangleright F\,A[\mathsf{p}])\\
&\mathsf{Tm}^\circ\,\Gamma^\circ\,A^\circ\,t && := \mathsf{Tm}\,(F\,\Gamma \triangleright \Gamma^\circ)\,(A^\circ[\mathsf{id},\, F\,t[\mathsf{p}]))\\
&\Gamma^\circ \triangleright^\circ A^\circ && := \Sigma(\Gamma^\circ[\mathsf{p} \circ F_{\triangleright.1}])(A^\circ[\mathsf{p} \circ F_{\triangleright.1} \circ \mathsf{p},\, \mathsf{q},\, \mathsf{q}[F_{\triangleright.1} \circ \mathsf{p}]])\\
&\ldots
\end{aligned}
$$

but which looks reasonable in sugary GLF notation:

$$
\begin{aligned}
&\mathsf{Con}^\circ\,\Gamma && := \mathsf{Ty}\,(\gamma : F\,\Gamma)\\
&\mathsf{Ty}^\circ\,\Gamma^\circ\,A && := \mathsf{Ty}\,(\gamma : F\,\Gamma,\, \gamma^\circ : \Gamma^\circ\,\gamma,\, \alpha : F\,A\,\gamma)\\
&\mathsf{Tm}^\circ\,\Gamma^\circ\,A^\circ\,t && := \mathsf{Tm}\,(\gamma : F\,\Gamma,\, \gamma^\circ : \Gamma^\circ\,\gamma)\,(A^\circ\,(\gamma,\, \gamma^\circ,\, F\,t\,\gamma))\\
&\Gamma^\circ \triangleright^\circ A^\circ && := \Lambda\,(F_{\triangleright.2}(\gamma,\, \alpha)).\,\Sigma(\gamma^\circ : \Gamma^\circ\,\gamma) \times A^\circ\,(\gamma,\, \gamma^\circ,\, \alpha)\\
&\ldots
\end{aligned}
$$

Sketch of the semantics. First, we give a short motivation. In the semantics, each PSh_i should be an universe of internal presheaves over an internal category. Clearly the semantics should involve categories, but there are well-known complications with the category of categories: a) there is no general Π type b) Π-types of presheaves and universes of presheaves are not stable under reindexing by arbitrary functors. The former issue could be addressed by having a "directed type theory", while the latter could be addressed with modalities. In the case of GLF we don't need either of these solutions. The reason is that we can't do any interesting categorical reasoning in GLF, and Base and In are used purely for managing internal/external languages, and it suffices to have enough semantic structure to represent the internal/external shifts.

The model of GLF is constructed in two steps. First, we give a model for the theory that has PSh, Base and In as sorts but does not support U, and then take presheaves over that model to obtain a model of a 2LTT where U represents the outer layer. In the inner model, we start with an inductive definition of certain *trees of categories*:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Tree}\,(B : \mathsf{Cat}) : \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{node} : (\Gamma : \mathsf{PSh}\,B)(n : \mathbb{N})(C : \mathsf{Fin}\,n \to \mathsf{Fib}\,(B \triangleright \mathsf{Disc}\,\Gamma))\\
&\qquad\quad \to ((i : \mathsf{Fin}\,n) \to \mathsf{Tree}\,(B \triangleright \mathsf{Disc}\,\Gamma \triangleright C\,i))\\
&\qquad\quad \to \mathsf{Tree}\,B
\end{aligned}
$$

Here, PSh means presheaves in sets, Fib is cartesian fibrations, Disc creates a discrete fibration from a presheaf and $-\triangleright-$ takes the total category of a fibration. Now, the objects of the semantic base category are elements of $\mathsf{Tree}\,1$, and morphisms between trees are level-wise natural transformations between the Γ components together with $\mathsf{Fin}\,n \to \mathsf{Fin}\,m$ renamings of subtree indices. The non-discrete Fib components are preserved by morphisms.

In a nutshell, each node represents a presheaf universe and each edge represents an internal/external switch. A semantic element of Base selects a node of a tree, while an In is an index that points to a subtree of a node. A semantic PSh is a dependent presheaf over a Γ in a given node. Extending a context with an In binding adds a new empty subtree to a given node, and extending a context with a presheaf variable extends the Γ presheaf in a node with a dependent presheaf.

References

[1] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *ArXiv e-prints*, may 2019.

[2] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019.

[3] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[4] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 204–213. IEEE Computer Society, 1999.

[5] Ambrus Kaposi, Simon Huber, and Christian Sattler. Gluing for type theory. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[6] Ambrus Kaposi and Szumi Xie. Second-order generalised algebraic theories: Signatures and first-order semantics. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, volume 299 of *LIPIcs*, pages 10:1–10:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[7] Ian Orton and Andrew M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rébecca Zucchini. Cocon: Computation in contextual type theory. *CoRR*, abs/1901.03378, 2019.

[9] Jonathan Sterling. *First Steps in Synthetic Tait Computability.* PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2021.

[10] Taichi Uemura. A general framework for the semantics of type theory. *CoRR*, abs/1904.04097, 2019.

A Curry-Howard correspondence for intuitionistic inquisitive logic

Ivo Pezlar[1] and Vít Punčochář[2]

[1] Czech Academy of Sciences, Institute of Philosophy Prague, Czechia
pezlar@flu.cas.cz
[2] Czech Academy of Sciences, Institute of Philosophy Prague, Czechia
puncochar@flu.cas.cz

Abstract

In this paper, we introduce a typed natural deduction system for propositional intuitionistic inquisitive logic. The term calculus we use to establish a Curry-Howard correspondence is lambda calculus extended with a new construct corresponding to the logical behaviour of the Split rule, a key rule of inquisitive logic. We show that the resulting system is normalizing. The existence of this system corroborates previous observations that questions have constructive content.

Extended abstract

Inquisitive logic [2] is a framework that accounts for both statements and questions within a unified formal system. In recent years, research on inquisitive logic has grown significantly and it has found its application in many other areas such as linguistics or philosophy of language [3].

Inquisitive logic is especially well-explored and understood from model-theoretic and algebraic points of view [12, 4]. Recently, there has been progress in the proof-theoretic understanding of the system [13, 7]. However, when it comes to a type-theoretic view, the picture of inquisitive logic becomes less clear. To our knowledge, this area has not yet been properly explored.

In this paper, we want to fill this gap and examine inquisitive logic from a type-theoretic point of view. The canonical system of inquisitive logic is based on classical logic of statements. However, in the type-theoretic context, it is natural to focus instead on intuitionistic inquisitive logic (InqIL), which is an inquisitive logic of questions based on intuitionistic logic of statements [5, 9, 10]. In particular, we introduce a Curry-Howard correspondence between a natural deduction system for propositional InqIL and a lambda calculus extended with a new construct select that will capture the logical behaviour of the key rule of inquisitive logic called Split.

The fact that this can be achieved shows that there is a close link between the notions of inquisitive/interrogative content and computational/constructive content. This was already partly hinted at by the following Ciardelli's observation:

> proofs [of dependencies; i.e., proofs where both premises and conclusion are questions] have an interesting kind of constructive content, reminiscent of the proofs-as-programs interpretation of intuitionistic logic: a proof of a dependency encodes a method for *computing* the dependency, i.e., for turning answers to the question premises into an answer to the question conclusion. ([2], p. 3; see also [1], p. 324)

This also raises a further intriguing question: can these approaches be combined? For example, while inquisitive semantics can model many kinds of questions, it is unsuitable for deductive or computational ones (e.g., "What is 2 + 2?" or "What follows from φ?") as its key semantic notion of informational support is closed under logical consequence, and thus computational

questions are trivially resolved by all information states. A type-theoretic approach, however, has tools for tackling these issues.

In the basic version of intuitionistic inquisitive logic, inquisitive disjunction is the primary question-forming operator and there is no primitive declarative disjunction. However, declarative disjunction can be added to the system, either in the form of the so-called tensor disjunction, as in [5, 10], or it can be defined as presupposition of the inquisitive disjunction, if we add to the language the presupposition modality $\circ$, as in [11]. We will also discuss these extensions.

We obtain InqIL by extending intuitionistic propositional logic (IPL) by the following rule known as Split:

$$\frac{\alpha \to (\varphi \vee \psi)}{(\alpha \to \varphi) \vee (\alpha \to \psi)}\ \text{Split}$$

where φ and ψ are arbitrary formulas corresponding to either questions or statements and α is a declarative, that is, a disjunction-free formula corresponding to a statement. The rule intuitively says that conditional questions are disjunctive questions resolved by suitable conditionals. In particular, the possible answers to the conditional question *whether q or r, if p* are, in accordance with Split, the conditionals *if p then q* and *if p then r*.

A key step in obtaining a Curry-Howard correspondence for InqIL rests on finding an appropriate constructive function that would capture the behaviour of the key Split rule from a computational point of view. To this end, we can utilize the results of [8] that introduced a generalized version of this rule with such a function in the context of a propositional fragment of Martin-Löf's constructive type theory.

We show that we can carry over this function into InqIL and use it to derive the Split rule and thus also provide its computational interpretation. The resulting lambda term capturing its behaviour will be as follows:

$$\frac{\dfrac{f : \alpha \to (\varphi \vee \psi) \qquad [x : \alpha]}{\mathsf{ap}(f, x) : \varphi \vee \psi} \qquad \dfrac{[y : \alpha \to \varphi]}{\mathsf{injl}(y) : (\alpha \to \varphi) \vee (\alpha \to \psi)} \qquad \dfrac{[z : \alpha \to \psi]}{\mathsf{injr}(z) : (\alpha \to \varphi) \vee (\alpha \to \psi)}}{\mathsf{select}(x.\mathsf{ap}(f, x), y.\mathsf{injl}(y), z.\mathsf{injr}(z)) : (\alpha \to \varphi) \vee (\alpha \to \psi)}$$

The crucial new construct select is defined as follows:

$$\frac{\begin{array}{ccc}[x : \alpha] & [y : \alpha \to \varphi] & [z : \alpha \to \psi] \\ c(x) : \varphi \vee \psi & d(y) : \chi & e(z) : \chi\end{array}}{\mathsf{select}(x.c(x), y.d(y), z.e(z)) : \chi}$$

with the following computation rules, where $t(x) : \varphi$ and $s(x) : \psi$:

$$\mathsf{select}(x.\mathsf{injl}(t(x)), y.d(y), z.e(z)) \Rightarrow d(\lambda x.t(x))$$
$$\mathsf{select}(x.\mathsf{injr}(s(x)), y.d(y), z.e(z)) \Rightarrow e(\lambda x.s(x))$$

Note that the new construct select is a variable-bining operator (the notation '$x.c(x)$' means that the variable x becomes bound in $c(x)$ by select) and that it is treated as an eliminatory noncanonical operator for disjunction (without appearances of the declarative formula α, the rule reduces to the standard disjunction elimination rule). From a functional perspective, adding select allows us to compute even open terms to canonical values, as long as the free variables of those open terms range over declarative formulas α only.

Furthermore, we consider a variant of InqIL called InqIL$^\circ$ extended with a presupposition modality $\circ$. This modality was introduced in [11] and defined via the following rules:

$$\frac{\varphi}{\circ\varphi}\ \circ\text{I} \qquad \frac{\circ\varphi \qquad \begin{array}{c}[\varphi]^i \\ \alpha\end{array}}{\alpha}\ \circ\text{E}_i$$

This modality is inspired by the truncation modality from homotopy type theory [6] that turns types into mere propositions, that is, types that are inhabited by at most one term (up to equivalence). The presupposition modality $\circ$ turns inquisitive formulas into declarative ones. As mentioned above, it can be used to define declarative disjunction as $\circ(\varphi \vee \psi)$. And, analogously to Split and select, we introduce new constructs for $\circ$I and $\circ$E called pre and sup that will allow us to extend the Curry-Howard correspondence to InqIL° as well.

Having a typed natural deduction system for inquisitive logic makes it possible to utilize Tait's computability method for proving normalization [14]. Specifically, as the notion of reduction $\Rightarrow$ that we will define via computation rules is a deterministic weak head reduction (that is, there is at most one possible reduction for any given term), we show that both InqIL and InqIL° are weakly normalizing. However, we suspect that if a more general notion of reduction is adopted, the strong normalization property can be obtained as well.

Finally, establishing a Curry-Howard correspondence for inquisitive logic sheds further light on the connection between the *formulas-as-types* principle innate to the Curry-Howard correspondence and the *questions-as-information types* interpretation of inquisitive logic [1]. First, our results confirm that formulas of inquisitive logic can indeed be regarded as types, as previously suggested by Ciardelli [1], p. 352 (see also [2], p. 110).

Second, the inherent distinction between information types and singleton types of inquisitive logic can be seen as paralleling the type-theoretic distinction between types and mere propositions. From this perspective, both truncation of type theory and presupposition of inquisitive logic can be seen as operators for suppressing content: computational one in the case of truncation and inquisitive one in the case of presupposition.

References

[1] Ivano Ciardelli. Questions as information types. *Synthese*, 195(1):321–365, 2018.

[2] Ivano Ciardelli. *Inquisitive Logic. Consequence and Inference in the Realm of Questions*. Springer, Cham, 2023.

[3] Ivano Ciardelli, J. A. G. Groenendijk, and Floris Roelofsen. Inquisitive semantics: A new notion of meaning. *Language and Linguistics Compass*, 7(9):459–476, 2013.

[4] Ivano Ciardelli, J. A. G. Groenendijk, and Floris Roelofsen. *Inquisitive Semantics*. Oxford University Press, Oxford, 2019.

[5] Rosalie Iemhoff, Fan Yang, and Ivano Ciardelli. Questions and dependency in intuitionistic logic. *The Notre Dame Journal of Formal Logic*, 61(1):75–115, 2020.

[6] Univalent Foundations of Mathematics. *Homotopy Type Theory*. Institute for Advanced Study, Princeton, 2013.

[7] Valentin Müller. On the Proof Theory of Inquisitive Logic, Master's thesis, University of Amsterdam, 2023.

[8] Ivo Pezlar. Constructive validity of a generalized Kreisel-Putnam rule. *Studia Logica*, online first, 2024.

[9] Vít Punčochář. A Generalization of inquisitive semantics. *Journal of Philosophical Logic*, 45(4):399–428, 2016.

[10] Vít Punčochář. Algebras of information states. *Journal of Logic and Computation*, 27(5):1643–1675, 2017.

[11] Vít Punčochář and Ivo Pezlar. Informative presupposition in inquisitive logic. In Agata Ciabattoni, David Gabelaia, and Igor Sedlár, editors, *Advances in Modal Logic, Volume 14*, pages 609–630, London, 2024. College Publications.

[12] Floris Roelofsen. Algebraic foundations for the semantic treatment of inquisitive content. *Synthese*, 190:79–102, 2013.

[13] Will Stafford. Proof-theoretic semantics and inquisitive logic. *Journal of Philosophical Logic*, 50(5):1199–1229, 2021.

[14] William Walker Tait. Intensional interpretations of functionals of finite type I. *Journal Symbolic Logic*, 32(2):198–212, 1967.

Y is not typable in λU

Herman Geuvers[1,2]*and Joep Verkoelen

[1] iCIS, Radboud University Nijmegen, The Netherlands
[2] Technical University Eindhoven, The Netherlands

Abstract

The type theories λU and λU^- are known to be logically inconsistent. For λU, this is known as Girard's paradox [7]; for λU^- the inconsistency was proved by Coquand [3]. It is also known that the inconsistency gives rise to a so called *looping combinator*: a family of terms L_n such that $L_n f$ is convertible with $f(L_{n+1} f)$. It is un-known whether a fixed point combinator exists in these systems. Hurkens [9] has given a simpler version of the paradox in λU^-, giving rise to an actual proof term that can be analyzed, and which is proven to be a looping combinator and not a fixed point combinator in [2]. However, the underlying untyped term is a real fixed point combinator.

Here we analyze the possibility of typing a fixed point combinator in λU and we prove that the Curry and Turing fixed point combinators Y and Θ cannot be typed in λU, and the same holds for Ω.

Although systems like $\lambda\star$ and λU are logically inconsistent, computationally they are still interesting, because not all terms are β-convertible. The first to study the computational power of these inconsistent systems was Howe [8], going back to earlier (unpublished) work of [10]. Howe coined the terminology *looping combinator* for a family of terms $\{L_n\}_{n\in\mathbb{N}}$ such that $L_n\, f =_\beta f(L_{n+1}\, f)$, and he showed that a looping combinator can be defined in $\lambda\star$. Using a looping combinator, it can be shown that the equational theory (the theory of β-conversion) is undecidable and that the theory is Turing complete.

When Girard [7] proved the paradox in 1972, he did that for λU, an extension of higher order logic with polymorphic domains and quantification over all domains. This system allows less type constructions than $\lambda\star$, but that has the advantage that it is somewhat easier to see what is going on. By that time, it was unclear whether λU^-: higher order logic with polymorphic domains (but no quantification over all domains) was inconsistent.

In 1994, Coquand [3] proved that λU^- is inconsistent as well, by encoding Reynolds' result [11], stating that no set-theoretic model of polymorphic lambda calculus exists, into λU^-. Later, Hurkens gave a considerably shorter proof [9], which is based on interpreting Russell's paradox in λU^-. Recently, Coquand [4] has given an adapted presentation of Hurkens' proof, emphasizing the relation with Reynolds' result.

Here we analyze the paradox in λU syntactically. (For a semantic analysis, relating the paradox to models of higher order logic, see [6].) The main question we are interested in is whether there exists a fixed-point combinator in λU. We give a partial answer by showing that the well-known Turing and Curry fixed-point combinators (Θ and Y) cannot be typed in λU.

We assume λU to be known (see [1, 5]), so we don't give the typing rules but we just emphasize that we divide the set of variables $\mathcal{V}$ into three disjoint sets $\mathrm{var}^\triangle$, $\mathrm{var}^\square$ and $\mathrm{var}^\star$ for which we use standard characters: $\mathrm{var}^\triangle = \{k_1, k_2, k_3, \ldots\}$, $\mathrm{var}^\square = \{\alpha, \beta, \gamma, \ldots\}$, $\mathrm{var}^\star = \{x, y, z, \ldots\}$. So a variable that lives in a type $A : \star$ is typically x, y or z etcetera. We also define the syntactical

*herman@cs.ru.nl

categories *Kinds* (K_1, K_2, K_3), *Constructors* (P, Q, R) and *Proof terms* (t, p, q) as follows.

$$\begin{array}{lll} \text{Kinds} & K & ::= \quad k \mid \star \mid K \to K \mid \Pi k{:}\Box.K \\ \text{Constructors} & P & ::= \quad \alpha \mid \lambda\alpha{:}K.P \mid PP \mid P \to P \mid \lambda k{:}\Box.P \mid PK \mid \Pi\alpha{:}K.P \\ \text{Proof terms} & t & ::= \quad x \mid \lambda x{:}P.t \mid tt \mid \lambda\alpha{:}K.t \mid tP \mid \lambda k{:}\Box.p \mid pK \end{array}$$

An important property of λU (which is not the case in $\lambda\star$) is that

Lemma 1. *All kinds and constructors of λU are strongly normalizing.*

Therefore, type checking is decidable in λU. For t a proof term of λU, we define the *erasure* of t, denoted by $|t|$, as follows, by induction on the construction of proof terms.

$$\begin{array}{lcll} |x| & = & x & \\ |\lambda x{:}P.p| & = & \lambda x.|p| & \text{if } P \in \text{Constructors} \\ |\lambda\alpha{:}K.p| & = & |p| & \text{if } K \in \text{Kinds} \\ |\lambda k{:}\Box.p| & = & |p| & \end{array} \qquad \begin{array}{lcll} |pq| & = & |p||q| & \text{if } p, q \in \text{Proof terms} \\ |pP| & = & |p| & \text{if } P \in \text{Constructors} \\ |pK| & = & |p| & \text{if } K \in \text{Kinds} \end{array}$$

We say that an untyped lambda term M is *typable in λU* iff there exist Γ, t, A such that $\Gamma \vdash t : A : \star$ and $|t| = M$. We prove the following result

Proposition 1. *The terms Ω, Y and Θ are not typable in λU.*

This result comes as a corollary of a more general result:

Theorem 2. Double self-application *is not possible in λU.*
Here we mean with "double self-application" a term $t : A : \star$ such that $|t| = (\lambda x.N)(\lambda y.P)$ and N contains a sub-term $x\,x$ and P contains a sub-term $y\,y$.

The Theorem is proving by analyzing the so called *parse tree* of a type, following ideas from [12]. The argument basically consists of two parts:

1. if t contains a self-application, so $|t|$ contains a sub-term $x\,x$, then the type of x in t is of the form $\Pi\vec{v} : \vec{V}.\alpha\vec{T} \to \ldots$ with $\alpha \in \vec{v}$;

2. if $|q| = \lambda y.N$ where N contains $y\,y$, then the type of q is not of the $\Pi\vec{v} : \vec{V}.\alpha\vec{T} \to \ldots$ with $\alpha \in \vec{v}$.

From this the Theorem follows.

If we now look back at the looping combinator L_0 that can be derived from the inconsistency proof of Hurkens [9], and we erase all type information, we obtain the following term.

$$|L_i| = L = \lambda f.(\lambda x.x(\lambda pq.f(qpq))x)(\lambda y.yy)$$

In the untyped λ-calculus, this is a fixed-point combinator and an interesting one, because it contains no *double self-application*, as Ω, Y and Θ do.

References

[1] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[2] G. Barthe and Th. Coquand. Remarks on the equational theory of non-normalizing pure type systems. *Journal of Functional Programming*, 16(2):137–155, 2006.

[3] Th. Coquand. A new paradox in type theory. In *Logic, Methodology and Philosophy of Science IX: Proc. Ninth Int. Congress of Logic, Methodology, and Philosophy of Science*, pages 7–14. Elsevier, 1994.

[4] Thierry Coquand. A variation of Reynolds-Hurkens paradox. In Venanzio Capretta, Robbert Krebbers, and Freek Wiedijk, editors, *Logics and Type Systems in Theory and Practice - Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday*, volume 14560 of *Lecture Notes in Computer Science*, pages 111–117. Springer, 2024.

[5] H. Geuvers. *Logics and Type Systems*. PhD thesis, Radboud University, Nijmegen, 1993.

[6] H. Geuvers. (In)consistency of extensions of higher order logic and type theory. In Th. Altenkirch and C. McBride, editors, *Types for Proofs and Programs Int. Workshop, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *LNCS*, pages 140–159. Springer, 2007.

[7] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[8] D. J. Howe. The computational behaviour of Girard's paradox. In *Proceedings of the 2nd Symposium on Logic in Computer Science*, pages 205–214. IEEE, 1987.

[9] A.J.C. Hurkens. A simplification of Girard's paradox. In *TLCA '95: Proceedings of the 2nd Int. Conf. on Typed Lambda Calculi and Applications*, pages 266–278, London, UK, 1995. Springer.

[10] M.B. Reinhold. Typechecking is undecidable when 'type' is a type, 1986.

[11] J.C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D.B. MacQueen, and G.D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984.

[12] P. Urzyczyn. Type reconstruction in Fω. *Mathematical Structures in Comp. Sci.*, 7(4):329–358, 1997.

A Ghost Sort for Proof-Relevant yet Erased Data in Rocq and MetaRocq

Johann Rosain[1], Matthieu Sozeau[1], and Théo Winterhalter[2]

[1] LS2N & Inria de l'Université de Rennes,
Nantes, France
[2] LMF & Inria Saclay,
Saclay, France

Abstract

We present an extension of Rocq's type theory and implementation with a new `Ghost` sort that models proof relevant data that is erased at extraction, inspired by recent work on ghost types [11], sort polymorphism [9] and certified erasure [3] in MetaRocq [10].

Rocq is based on a dependent type theory with two main kinds of universes, the `Type` predicative universes that contain computationally relevant data and the `Prop`/`SProp` impredicative universes that represent irrelevant data that can be erased at extraction to produce (relatively) efficient programs from Rocq definitions. The initial work on extraction in Rocq was done by Paulin-Mohring [8] during her PhD and followed up by the PhD of Letouzey [5], which gives an accurate account of the current state of extraction in Rocq. Technically, extraction is based on elimination restrictions that ensure that propositional content (in `Prop` or `SProp`) cannot be used in a relevant way to produce computational data (with a type in the `Type` universe), with only one exception: so-called sub-singleton propositions can be inspected when producing computational data (e.g. natural numbers). The trivial (`True`) and absurd (`False`) propositions, the conjunction of propositions (`A ∧ B`), the propositional equality type (`eq`) and the accessibility predicate (`Acc`) all fall into this criterion: they have at most one constructor, whose arguments (if any) are all propositional. In contrast, disjunction (`A ∨ B`) and existential quantification (`∃ x : nat. P`) fall outside this criterion, with good reason: if one could write a program that takes a disjunction proof `A ∨ B` and produces 0 in case the proof comes from the left branch and returns 1 otherwise, then proofs would have to be kept during extraction! Forster et al. [3] provide a formal proof on top of MetaRocq that the process of extraction based on elimination restrictions preserves semantics, while erasing all propositional content and type annotations. However, there are cases where data that should be erased remains in extracted terms.

The case of Accessibility. Singleton elimination of `Prop` applies to accessibility:

```
Inductive Acc {A} (R : relation A) (x : A) : Prop :=
| Acc_intro : (∀ y, R y x → Acc R y) → Acc R x.
```

```
Definition well_founded {A} R :=
  ∀ x, Acc R x.
```

This inductive type represents accessibility proofs of relations, and is used to define constructive well-foundedness of relations. Thanks to singleton elimination, this allows to derive principles of *well-founded* (a.k.a. Noetherian) recursion and induction for `Type`-valued predicates:

```
Definition nat_lt_ind (P : nat → Type) : (∀ n, (∀ m, m < n → P m) → P n) → ∀ n, P n :=
  ... Acc_rect ...
```

The justification to erase uses of `Acc` during extraction is a bit more involved than for other (sub)singletons: indeed as `Acc` is a recursive type, we need to ensure that definitions in Rocq cannot distinguish between two accessibility proofs of potentially different depths. The correctness theorem of extraction crucially holds only on *closed* Rocq terms, for which we can

assume by the canonicity property of the theory that any fixed-point definition on `Acc` proofs will be able to consume as many `Acc_intro` constructors as necessary to reach a normal form. This is however not the case in Rocq itself, where definitional equality is checked on *open* terms and the theory *does* distinguish between accessibility proofs, e.g. one that is a variable and one that starts with a constructor cannot be considered definitionally equal, otherwise non-termination can ensue. Putting `Acc` in `Prop` while still considering it a (sub)singleton hence goes against a definitional proof irrelevance interpretation of `Prop`, even if the weaker principle of propositional proof irrelevance in `Prop` can be added consistently to Rocq.

So, in the definitionally proof-irrelevant sort `SProp` in Rocq, accessibility is restricted to eliminations to `SProp` only.[1] This is an unfortunate situation, as this precludes using `SProp` as a replacement for `Prop` in many situations. A better compromise is to recognize that accessibility is definitionally proof-relevant (so cannot live in `SProp`) but still erasable: this is one purpose of the new `Ghost` sort we introduce.

No ghost data. Currently, as extraction is based on the `Prop`/`Type` separation, it is impossible to erase proof-relevant data at extraction. A typical use-case appears when using indexed or dependent data types in programs, e.g. finite numbers:

```
Inductive fin : nat → Set :=
| fin0 {n} : fin (S n)
| finS {n} : fin n → fin (S n).

Fixpoint lookup {A} (l : list A) : fin (length l) → A :=
  match l return fin (length l) → A with
  | nil ⇒ fun (f : fin 0) ⇒
     False_rect (match f in (fin 0) with end)
  | cons a l ⇒ fun f ⇒ match f with
                       | fin0 ⇒ a
                       | finS f' ⇒ lookup l f'
                       end
  end.
```

```
(* Extraction to OCaml: *)
type fin =
| Fin0 of nat
| FinS of nat * fin

let rec lookup l f =
 match l with
 | Nil → assert false
 | Cons (a, l0) →
    begin
     match f with
     | Fin0 _ → a
     | FinS (_, f') → lookup l0 f'
    end
```

In the computation of the `lookup` function, the indices of the `fin` structure are never inspected. They are only used to logically justify that the first branch is unreachable. Extraction is unaware of that intention[2] and carries around all the index data of `fin`, as witnessed by the way `fin` is extracted to OCaml.

Here, we would like to state in `fin`'s type declaration that the natural number cannot be used computationally, but only in ghost contexts, so that Rocq can enforce that the `lookup` function does not inspect unduly these indices and that they hence can be erased during extraction.

Progress on this subject was made recently by Winterhalter [11], who introduced Ghost Type Theory (GTT), an extension of dependent type theory with a new sort for ghost types and values. This theory allows marking indices such as the `fin` index as ghost data and ensures the extraction property we expect. However, the meta-theory of GTT is not yet complete, and it shares the same defect as the `SProp` sort, being unable to accommodate a useful accessibility relation. We simplify this proposal, drawing inspiration from the system of Keller and Lasson [4] which already introduced a distinction between two predicative hierarchies named `Set` and `Type` which played the roles of non-erasable and erasable proof-relevant types. That system was designed with parametricity in mind though, not extraction.

[1] Lean deliberately ignores this issue and breaks transitivity of conversion due to that choice.
[2] One could in fact write functions that *do* inspect the indices.

Our work. Thanks to Rocq's recent support of sort polymorphism [9] and the new implementation of sort elimination constraints (see the companion abstract), we can experiment with different elimination rules for the `Ghost` sort. We are considering the following designs:

1. A term of a `Ghost` type can only be eliminated (i.e. in a `match`) to produce ghost content, except for $\bot$ which always has large elimination. Moreover, `Fixpoint` elimination is allowed from `Ghost` into any sort.

2. The `Ghost` sort can be eliminated into itself and `SProp` (which enables large elimination for $\bot$ by transitivity). `Fixpoint` elimination is also allowed from `Ghost` into any sort.

3. We restrict elimination to avoid creating computational content (i.e. we disable elimination of `Ghost` into `Type`), but always allow it for other sorts.

All of those enjoy the same desired property: they accommodate the accessibility predicate. Moreover, introducing a new sort allows one to add specific annotations for proof-relevant content to be erased at extraction — here, it would simply be to use the ghost version of the types/functions. For instance, it becomes possible to define a `lookup` function where extraction leads to a code that totally discards `fin`'s indices:

```
Inductive fin : nat@{Ghost|} → Set :=
| fin0 {n} : fin (S n)
| finS {n} : fin n → fin (S n).

Fixpoint lookup {A} (l : list A)
 : fin (length@{Ghost} l) → A :=
 match l return fin (length@{Ghost} l) → A with
 | nil ⇒ fun (f : fin 0) ⇒
    False_rect (match f in (fin 0) with end)
 | cons a l ⇒ fun f ⇒ match f with
                | fin0 ⇒ a
                | finS f' ⇒ lookup l f'
                end
 end.
```

```
(* Extraction to OCaml: *)
type fin =
| Fin0
| FinS of fin

let rec lookup l f =
 match l with
 | Nil → assert false
 | Cons (a, l0) →
    begin
     match f with
     | Fin0 → a
     | FinS f' → lookup l0 f'
    end
```

We are also considering replacing the equality of `Prop` by one in `Ghost`, using a specific `cast` term *à la* GTT. This would effectively get rid of the need of singleton elimination, and maybe of the `Prop` sort. This approach might fix the long-standing extraction inconsistency in the presence of univalence. Indeed, one can show that the equality in `Prop` and in `Type` are equivalent, meaning, from negation, one gets an equality $\mathbb{B} = \mathbb{B}$ in `Prop`, transporting `true` along that equality will yield an element that is provably equal to `false` and yet extracts to `true`. Choosing one of the first two designs would remove the extraction inconsistency, as in either of these cases, `Ghost` equality is not equivalent to `Type` equality.

While our initial experiments are made in Rocq, we plan to develop a MetaRocq proof to show the erasure theorem [10] when using `Ghost` extraction, and to make sure we do not break any meta-theoretical properties already proven about Rocq.

In this talk, we will present the `Ghost` sort through the development of examples, focusing on the treatment of accessibility and indexed data types, and summarize the status of the MetaRocq formalization. We will also compare our new `Ghost` sort with recent developments of quantitative type theories [7, 1], the type theory internalizing reasoning on non-interference of Liu et al. [6] and its logical properties with the ones of the erasure modality [2] (we expect to satisfy the same properties).

References

[1] Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 56–65, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355834. doi: 10.1145/3209108.3209189. URL https://doi.org/10.1145/3209108.3209189.

[2] Nils Anders Danielsson. Logical properties of a modality for erasure. 2019. URL https://www.cse.chalmers.se/~nad/publications/danielsson-erased.html.

[3] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656379.

[4] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Patrick Cégielski and Arnaud Durand, editors, *CSL*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. ISBN 978-3-939897-42-2.

[5] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004. URL http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf.

[6] Yiyun Liu, Jonathan Chan, Jessica Shi, and Stephanie Weirich. Internalizing Indistinguishability with Dependent Types. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi: 10.1145/3632886. URL https://doi.org/10.1145/3632886.

[7] Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi: 10.1007/978-3-319-30936-1_12. URL https://doi.org/10.1007/978-3-319-30936-1_12.

[8] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions. (Program Extraction in the Calculus of Constructions)*. PhD thesis, Paris Diderot University, France, 1989. URL https://tel.archives-ouvertes.fr/tel-00431825.

[9] Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. All Your Base Are Belong to $\mathcal{U}^s$: Sort Polymorphism for Proof Assistants. *Proc. ACM Program. Lang.*, 9(POPL):2253–2281, 2025. doi: 10.1145/3704912. URL https://doi.org/10.1145/3704912.

[10] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. To appear in Journal of the ACM, April 2023. URL https://inria.hal.science/hal-04077552.

[11] Théo Winterhalter. Dependent Ghosts Have a Reflection for Free. *Proc. ACM Program. Lang.*, 8(ICFP):630–658, 2024. doi: 10.1145/3674647. URL https://doi.org/10.1145/3674647.

A Fully Dependent Assembly Language

Yulong Huang and Jeremy Yallop

University of Cambridge, UK

We report our progress in developing an assembly language with fully dependent types to support type-preserving compilation of dependent type theory, and thus improving the compilation of existing systems which erase types at an early stage. In this abstract, we outline the fundamental design principles of the language and present our encoding of it in Agda.[1]

A type-preserving compiler [8] transforms the source code into a strongly typed assembly language with a series of typed compiler transformations, preserving types through all phases. Many optimizations benefit from type information. With dependent types, type signatures can contain code specifications and type-checking the assembly language verifies that the specifications are met, essentially implementing proof-carrying code [9].

Earlier attempts towards dependent assembly languages, such as Ritter's categorical abstract machine [10] and Winwood's Singleton [11], are incompatible with recent advances in dependently-typed transformations like CPS [3], ANF [6], closure conversion [2], and defunctionalization [4]. A desirable target language should both accommodate fully dependent types and be able to compile from the intermediate representations of previous transformations.

We propose a two-level design. The syntax of the typed assembly language consists of a set of instructions of a stack machine (I) and a dependently typed calculus of specifications (e, A).

$$I ::= \texttt{LIT}\ c \mid \texttt{APP} \mid \texttt{CLO}\ n\ lab \mid \texttt{POP} \mid I;I' \mid \ldots$$
$$e, A ::= x \mid e\,e' \mid lab\,\{e_1,\ \ldots,\ e_n\} \mid \Pi x{:}A.B \mid U \mid \ldots$$

The type judgement for each instruction *models* the computation like an abstract interpreter, in the form of $\Gamma \vdash I : \sigma \to \sigma'$. A stack σ is a list of specification terms. The judgement states that instruction I transforms stack σ to σ', similar to the conventional stack typing (e.g. WebAssembly [1]), but it keeps track of the stack's *content* instead of the types of the contents. For example, the rule for pushing a literal constant c in our language is $\Gamma \vdash \texttt{LIT}\ c : \sigma \to \sigma :: c$.

Tracking stack contents is necessary to support fully dependent types (since computation happens at type level), and separating specification from instructions avoids problems such as the need to check equality of instruction sequences during type checking. Function application, which is complicated by type dependency, is simply described by an application in the specification calculus (shown at the left below, note that B does not appear in the specification).

$$\frac{\Gamma \vdash e : \Pi x{:}A.B \qquad \Gamma \vdash e' : A}{\Gamma \vdash \texttt{APP} : \sigma :: e :: e' \to \sigma :: e\,e'} \qquad \frac{lab(\Delta, x{:}A \mapsto e : B) \in \Gamma \qquad \Gamma \vdash e_1, \ldots, e_n : \Delta}{\Gamma \vdash \texttt{CLO}\ n\ lab : \sigma :: e_1 :: \ldots :: e_n \to \sigma :: lab\,\{e_1,\ \ldots,\ e_n\}}$$

The specification language is a *defunctionalized* calculus, an intermediate representation from dependently typed defunctionalization [4]. It has no lambda abstractions and no way to create new functions. We can only create a closure of type $\Pi x{:}A.B$ by paring a labelled code lab (from a fixed set of labels, defined in the context Γ) with a list of terms $e_1, \ldots, e_n$ (of types Δ) instantiating the free variables in the code. In assembly, it is reflected by an operation $\texttt{CLO}\ n\ lab$ that takes the top n elements of the stack and forms a closure object with the label lab on top of the stack, as shown in the judgement at the right above.

The specification calculus is shown to be consistent [4]. We need to prove type safety for the assembly, in other words, that the typing rule's abstract interpretation correctly models runtime behaviour.

[1] See https://github.com/H-Yulong/ShallowStack.

We are in the process of encoding the assembly language in Agda to formally verify its meta-theory. The encoding is challenging: besides the classic problem of encoding dependent type theory in itself, we also have to find a suitable representation for labelled defunctionalized code.

We use shallow embedding [7, 5] to encode syntax, which coincides the definitional equality of the object theory and the host language to avoid the problem of having tedious equality conversions (known as "transport hell") — every equation in the object theory is trivially resolved to `refl` in the host.

Defunctionalization is trickier. As we have previously observed [4], the usual method of representing each function label with a constructor of a GADT fails to extend to inductive families, because it requires the data type to be indexed by itself (hence failing positivity checks), and the interpretation function for code labels is not obviously terminating.[2]

We have found a workaround for the positivity checks to define an indexed family `Pi` over the shallow-embedded syntax for defunctionalized code and an interpretation function `interp` that passes Agda's termination checker, and we will present the techniques used in the talk. Below shows the (simplified) type signatures of `Pi` and `interp`, and code labels for two functions.[3]

```
data Pi : (id : ℕ) (Γ : Con) (A : Ty Γ) (B : Ty (Γ , A)) → Set where
  CNat : Pi 0 · Nat U · ⊢ λx:Nat.Nat
  App : Pi 0 (·, U, Π 0 U, Π 1 (app 1 0)) 2 (app 2 0) A : U, B : ΠA:U.U, f : Πx:A.B ⊢ λx:A.f x
interp : Pi id Γ A B → Tm (Γ , A) B
```

The intrinsic typing rules of the stack machine are straightforward to express with shallow-embedded syntax and defunctionalization. The rules for `LIT` and `APP` are exactly like their typing judgements. Instruction `CLO` n *lab* takes an instance argument `pf` which proves that the first n items on the stack have types Δ, and Agda can find this proof automatically. We also have rules for creating and eliminating base types such as booleans and natural numbers.

```
data Instr (Γ : Con) : Stack Γ m → Stack Γ n → Set where
  LIT : (n : ℕ) → Instr Γ σ (σ :: nat n)
  APP : {f : Tm Γ (Π A B)} {a : Tm Γ A} → Instr Γ (σ :: f :: a) (σ :: app f a)
  CLO : (n : ℕ)(lab : Pi id Δ A B)
        {{ pf : Γ ⊢ (take n σ) of Δ }} → Instr Γ (drop n σ :: lab [[ pf ]]s)
```

Here is a type-checked example of an instruction sequence that computes 5 via `App` and `Add` (which corresponds to $\lambda y{:}Nat.x + y$). Firstly, it creates a closure for `App`, instantiating the free variables to Nat, `CNat{}`, and `Add{2}`, then applies the closure to 3, eventually computing $2+3$. The assembly code type-checks because $2 + 3 = 5$, which Agda can realize without proofs.

```
code : Is · · (· :: nat 5)                  Abstract stack content
code = TLIT Nat                             Nat
    >> CLO 0 CNat                           Nat :: CNat{}
    >> LIT 2                                Nat :: CNat{} :: 2
    >> CLO 1 Add                            Nat :: CNat{} :: Add{2}
    >> CLO 3 App                            App{Nat, CNat{}, Add{2}}
    >> LIT 3                                App{Nat, CNat{}, Add{2}} :: 3
    >> APP                                  5
    >> RET                                  5
```

Our dependent assembly language uses a two-level design that leaves the machine's model simple, but maintains a rich and expressive type system to accommodate type-preserving compilation. We plan to use this Agda formalization to investigate type erasure, assembly code generation and optimization, and incorporating quantitative type theory in the future.

[2] We use type-in-type in this abstract and omitted a problem with universe levels for simplicity.

[3] We use de-Bruijn indices (bold numbers) for variables.

References

[1] WebAssembly Core Specification.

[2] William J. Bowman and Amal Ahmed. Typed closure conversion for the calculus of constructions. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 797–811. ACM, 2018.

[3] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. Type-preserving CPS translation of Σ and Π types is not not possible. *Proc. ACM Program. Lang.*, 2(POPL):22:1–22:33, 2018.

[4] Yulong Huang and Jeremy Yallop. Defunctionalization with dependent types. *Proc. ACM Program. Lang.*, 7(PLDI):516–538, 2023.

[5] Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory is morally correct. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 329–365. Springer, 2019.

[6] Paulette Koronkevich, Ramon Rakow, Amal Ahmed, and William J. Bowman. ANF preserves dependent types up to extensional equality. *J. Funct. Program.*, 32:e12, 2022.

[7] Conor McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In Bruno C. d. S. Oliveira and Marcin Zalewski, editors, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010*, pages 1–12. ACM, 2010.

[8] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 85–97. ACM, 1998.

[9] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997.

[10] Eike Ritter. Categorical abstract machines for higher-order typed lambda-calculi. *Theor. Comput. Sci.*, 136(1):125–162, 1994.

[11] Simon Winwood and Manuel M. T. Chakravarty. Singleton: a general-purpose dependently-typed assembly language. In Stephanie Weirich and Derek Dreyer, editors, *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, pages 3–14. ACM, 2011.

Efficient Program Extraction in Elementary Number Theory using the Proof Assistant Minlog

Franziskus Wiesnet[1]

Vienna University of Technology, Vienna, Austria, European Union
franziskus.wiesnet@tuwien.ac.at

Abstract

Overview We examine theorems of elementary number theory using a constructive, computational, and proof-theoretic approach. Key theorems include Euclid's proof of the infinitude of prime numbers, Bézout's identity, the Fundamental Theorem of Arithmetic, and Fermat's factorization method. These theorems have been formalized within the proof assistant Minlog, providing rigorous formal verification. Minlog allows the extraction of executable Haskell programs from proofs, demonstrating their computational utility. The implementations can be found in the folder `examples/arith` of the Minlog directory. Currently these files are only available and functional in the dev branch of Minlog. Instructions on how to install Minlog and switch to the dev branch can be found on the Minlog website [14]. All relevant files are also available in a GitHub repository (https://github.com/FranziskusWiesnet/MinlogArith/).

We explore interesting and instructive aspects of the proofs, the implementation in Minlog, and the extracted Haskell term. Additionally, we will highlight the unique features and advantages of Minlog as a proof assistant, offering the audience a deeper insight into its capabilities. The main focus of the presentation will be on the Fundamental Theorem of Arithmetic and Fermat's factorization method. The second one can be extended to the quadratic sieve, which is a sub-exponential factorization method. An implementation of the quadratic sieve in Minlog is the subject of future research.

While prior knowledge of Minlog, Haskell, or other programming languages can be helpful, it is not required, as we will provide all necessary explanations.

Innovations and Methodology While Minlog has primarily been used for the formalization of analysis, this study serves as a case study demonstrating that Minlog is also highly suitable for algorithmic number theory. The only article dealing with number theory (in this case, the greatest common divisor) in Minlog is [6]. However, that work primarily focused on the extraction of programs from classical proofs. Therefore, this new implementation can serve as a solid foundation for the formalization of number theory in Minlog.

Another innovation is that we have considered the efficiency of the extracted term right from the formulation of theorems and their proofs. Therefore, the proofs were formalized in two different versions. The first version is based on unary natural numbers, defined by 0 and the successor function. In contrast, the second version is based on positive binary numbers given by 1 and two successor functions, one appending 0 and the other appending 1. While the first version focuses primarily on the simplicity of the theorems and their proofs, the second version prioritizes the efficiency of the extracted term. Hence, we demonstrate how formulating theorems and structuring their proofs influence the efficiency of the extracted algorithms, bridging the gap between formal verification and practical applications. Furthermore, extracting proofs as Haskell programs (rather than as terms in Minlog) will also contribute to shortening the runtime of the extracted algorithms.

Background Minlog is a proof assistant developed in the 1990s by the logic group at the Ludwig-Maximilians-Universität München under the direction of Helmut Schwichtenberg [4, 13, 14, 15, 20, 21]. Several introductions to Minlog are available [31, 32, 33]. Recently, Minlog has predominantly been employed in the field of constructive analysis [5, 11, 16, 23, 26, 35]. However, there exists a broad spectrum of proofs in diverse domains that have been implemented in Minlog [2, 8, 22, 24, 25]. All of them share the characteristic that an implementation in Minlog not only focuses on the correctness of the proof but also on extracting programs from the proofs. As the primary goal of Minlog is to interpret proofs as programs and to work with them accordingly [1], it includes all the functions for the formal program extraction from proofs.

Minlog and the formal program extraction from proofs are based on an extension of HA^ω called *Theory of Computable Functionals* (TCF) as meta-theory. Its origins go back to Scott's seminal work on *Logic of Computable Functionals* [28] and Platek's PhD thesis [18], and it incorporates basic concepts introduced by Kreisel [10] and Troelsta [30]. In recent years the Munich team have made important progress in the development of TCF [3, 7, 9, 17, 19]. It is based on partial continuous functionals and information systems, serving as their designated model, known as *Scott model* [12, 27].

Illustrative examples. The **natural numbers** $\mathbb{N}$ are given by the constructors

$$0 : \mathbb{N}, \quad \mathrm{S} : \mathbb{N} \to \mathbb{N},$$

and the **positive binary numbers** $\mathbb{P}$ are given by

$$1 : \mathbb{P}, \quad \mathrm{S}_0 : \mathbb{P} \to \mathbb{P}, \quad \mathrm{S}_1 : \mathbb{P} \to \mathbb{P}.$$

Note that the binary representation is reversed. For example, $\mathrm{S}_0\,\mathrm{S}_1\,1$ represents the number 6, which corresponds to binary 110. The reason for starting with 1 instead of 0 is primarily to ensure that each number is uniquely determined by its constructors. Otherwise, for example 0 and $\mathrm{S}_0\,0$ would represent the same number, violating this uniqueness.

Already the definition of the **greatest common divisor** clearly illustrates the computational difference between the number types. On natural numbers, the greatest common divisor can be defined by the euclidean algorithm:

$$\begin{aligned} \gcd(0,n) &:= \gcd(n,0) := n \\ \gcd(\mathrm{S}\,m, \mathrm{S}\,n) &:= \begin{cases} \gcd(\mathrm{S}\,m, n-m) & \text{if } m < n \\ \gcd(m-n, \mathrm{S}\,n) & \text{otherwise} \end{cases} \end{aligned}$$

It should be noted that using division with remainder on the natural numbers would not make this algorithm more efficient, because the computation of division with remainder results essentially in an iterated application of the step case in the definition above.

On positive binary numbers, however, the greatest common divisor can be defined much more efficiently by *Stein's algorithm* (named after Josef Stein [29]):

$$\begin{aligned} \gcd(1,p) := \gcd(\mathrm{S}_1\,p, 1) := \gcd(\mathrm{S}_0\,p, 1) &:= 1 \\ \gcd(\mathrm{S}_0\,p, \mathrm{S}_0\,q) &:= \mathrm{S}_0(\gcd(p,q)) \\ \gcd(\mathrm{S}_0\,p, \mathrm{S}_1\,q) := \gcd(\mathrm{S}_1\,p, \mathrm{S}_0\,q) &:= \gcd(p, \mathrm{S}_1\,q) \\ \gcd(\mathrm{S}_1\,p, \mathrm{S}_1\,q) &:= \begin{cases} \gcd(\mathrm{S}_1\,p, q-p) & \text{if } p < q \\ \gcd(p-q, \mathrm{S}_1\,q) & \text{if } q < p \\ \mathrm{S}_1\,p & \text{otherwise.} \end{cases} \end{aligned}$$

One can see that in Stein's algorithm, at each step, at least one argument is reduced by one digit and is therefore at least halved. Noting that the subtraction of two binary numbers has approximately linear runtime in the number of digits, Stein's algorithm therefore has quadratic runtime in the number of digits. In contrast, the Euclidean algorithm on natural numbers has quadratic runtime only with respect to the absolute size of the arguments.

Bézout's identity is known as the statement that the greatest common divisor is a linear combination of the two arguments. That is, for integers a, b there are integers u, v with $\gcd(a,b) = u \cdot a + v \cdot b$. During the formalization, we wanted to remain within the respective number system, so the theorem had to be reformulated as

$$\forall_{n,m} \exists_{l_0} \exists_{l_1} . \quad \gcd(n,m) + l_0 \cdot n = l_1 \cdot m \quad \vee \quad \gcd(n,m) + l_0 \cdot m = l_1 \cdot n$$

for natural numbers, and as

$$\begin{aligned} \forall_{p_0,p_1} . \quad & \exists_q \; q \cdot p_0 = p_1 \quad \vee \quad \exists_q \; q \cdot p_1 = p_0 \\ \vee \quad & \exists_{q_0} \exists_{q_1} \; \gcd(p_0,p_1) + q_0 \cdot p_0 = q_1 \cdot p_1 \\ \vee \quad & \exists_{q_0} \exists_{q_1} \; \gcd(p_0,p_1) + q_1 \cdot p_1 = q_0 \cdot p_0 \end{aligned}$$

for positive binary numbers. Note that, since integers in Minlog are defined as a type sum of negative binary numbers, zero, and positive binary numbers, the case distinction would be implicitly present in the proof anyway.

The proofs were carried out by induction over an upper bound of the numbers, which is seen as a natural number. In the induction step itself, a case distinction was made based on the structure of the numbers. Specifically for positive binary numbers, this means that the statement

$$\begin{aligned} \forall_{l,p_0,p_1} . \quad & p_0 + p_1 < l \quad \to \\ & \exists_q \; q \cdot p_0 = p_1 \quad \vee \quad \exists_q \; q \cdot p_1 = p_0 \\ \vee \quad & \exists_{q_0} \exists_{q_1} \; \gcd(p_0,p_1) + q_0 \cdot p_0 = q_1 \cdot p_1 \\ \vee \quad & \exists_{q_0} \exists_{q_1} \; \gcd(p_0,p_1) + q_1 \cdot p_1 = q_0 \cdot p_0 \end{aligned}$$

was proven by induction on $l : \mathbb{N}$ and case distinction on $p_0 \in \{1, \mathrm{S}_0\, q_0, \mathrm{S}_1\, q_0\}$ and $p_1 \in \{1, \mathrm{S}_0\, q_1, \mathrm{S}_1\, q_1\}$. In particular, properties of both natural numbers and positive binary numbers were combined. The individual cases are then quite straightforward, but in many instances, a further case distinction according to the four cases in the statement is required.

A similar approach was also chosen for the Fundamental Theorem of Arithmetic and Fermat's factorization method. A detailed presentation can be found in [34].

Acknowledgements. The research for this document was funded by the Austrian Science Fund (FWF) **10.55776/ESP576**.

Many thanks to *Helmut Schwichtenberg* for providing valuable tips on writing the Minlog code and for integrating it into the official Minlog version.

I am grateful to the anonymous reviewers for their thoughtful suggestions. Although space and time constraints prevented me from including all of them, their input was highly appreciated.

References

[1] Holger Benl and Helmut Schwichtenberg. Formal Correctness Proofs of Functional Programs: Dijkstra's Algorithm, a Case Study. In U. Berger and H. Schwichtenberg, editors, *Computational*

Logic, volume 165 of *Series F: Computer and Systems Sciences*, pages 113–126. Proceedings of the NATO Advanced Study Institute on Computational Logic, held in Marktoberdorf, Germany, July 29 – August 10, 1997, Springer Berlin Heidelberg, 1999.

[2] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program Extraction from Normalization Proofs. *Studia Logica*, 82(1):25–49, February 2006.

[3] Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114(1–3):3–25, April 2002.

[4] Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Monika Seisenberger. *Minlog - A Tool for Program Extraction Supporting Algebras and Coalgebras*, pages 393–399. Springer Berlin Heidelberg, 2011. 4th International Conference, CALCO 2011, Winchester, UK, August 30 – September 2, 2011. Proceedings.

[5] Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Hideki Tsuiki. Logic for Gray-code Computation. In D. Probst and P. Schuster, editors, *Concepts of Proof in Mathematics, Philosophy, and Computer Science*, pages 69–110. De Gruyter, July 2016.

[6] Ulrich Berger and Helmut Schwichtenberg. The greatest common divisor: A case study for program extraction from classical proofs. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, volume 1158, pages 36–46. Springer Berlin Heidelberg, 1996. International Workshop, TYPES '95 Torino, Italy, June 5–8, 1995 Selected Papers.

[7] Simon Huber, Basil A. Karádais, and Helmut Schwichtenberg. Towards a Formal Theory of Computability. In R. Schindler, editor, *Ways of Proof Theory: Festschrift for W. Pohlers*, pages 257–282. De Gruyter, December 2010.

[8] Hajime Ishihara and Helmut Schwichtenberg. Embedding classical in minimal implicational logic. *Mathematical Logic Quarterly*, 62(1–2):94–101, January 2016.

[9] Basil A. Karádais. *Towards an Arithmetic for Partial Computable Functionals*. PhD thesis, Ludwig-Maximilians University Munich, 2013.

[10] Georg Kreisel. Interpretation of Analysis by Means of Constructive Functionals of Finite Types. In A. Heyting, editor, *Constructivity in mathematics*, pages 101–128. North-Holland Pub. Co., 1959.

[11] Nils Köpp and Helmut Schwichtenberg. Lookahead analysis in exact real arithmetic with logical methods. *Theoretical Computer Science*, 943:171–186, January 2023.

[12] Kim Guldstrand Larsen and Glynn Winskel. Using information systems to solve recursive domain equations. *Information and Computation*, 91(2):232–258, April 1991.

[13] Kenji Miyamoto. *Program extraction from coinductive proofs and its application to exact real arithmetic*. PhD thesis, Ludwig-Maximilians University Munich, 2013.

[14] Kenji Miyamoto. The Minlog System. https://www.mathematik.uni-muenchen.de/~logik/minlog/, 2024.

[15] Kenji Miyamoto, Fredrik Nordvall Forsberg, and Helmut Schwichtenberg. Program Extraction from Nested Definitions. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 370–385. Springer Berlin Heidelberg, 2013. 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings.

[16] Kenji Miyamoto and Helmut Schwichtenberg. Program extraction in exact real arithmetic. *Mathematical Structures in Computer Science*, 25(8):1692–1704, November 2014.

[17] Iosif Petrakis. Advances in the Theory of ComputableFunctionals TCF+ due to its Implementation, 2013. online: https://www.math.lmu.de/~petrakis/TCF+.pdf.

[18] Richard Alan Platek. *Foundations of recursion theory*. PhD thesis, Stanford University, 1966.

[19] Helmut Schwichtenberg. Primitive Recursion on the Partial Continuous Functionals. In M. Broy, editor, *Informatik und Mathematik*, pages 251–268. Springer Berlin Heidelberg, 1991.

[20] Helmut Schwichtenberg. Proofs as Programs. In P. Aczel, H. Simmons, and S. Wainer, editors, *Proof Theory: A selection of papers from the Leeds Proof Theory Programme 1990*, page 79–114,

Cambridge, 1993. Cambridge University Press. Title from publisher's bibliographic system (viewed on 24 Feb 2016).

[21] Helmut Schwichtenberg. Minlog. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600, pages 151–157. Springer Berlin Heidelberg, 2006.

[22] Helmut Schwichtenberg. Program Extraction from Proofs: The Fan Theorem for Uniformly Coconvex Bars. In S. Centrone, S. Negri, D. Sarikaya, and P. Schuster, editors, *Mathesis Universalis, Computability and Proof*, volume 412 of *Synthese Library*, pages 333–341. Springer International Publishing, 2019.

[23] Helmut Schwichtenberg. Logic for Exact Real Arithmetic: Multiplication. In *Mathematics for Computation (M4C)*, chapter 3, pages 39–69. World Scientific, April 2023.

[24] Helmut Schwichtenberg, Monika Seisenberger, and Franziskus Wiesnet. Higman's Lemma and its Computational Content. In R. Kahle, T. Strahm, and T. Studer, editors, *Advances in Proof Theory*, pages 353–375. Springer International Publishing, 2016.

[25] Helmut Schwichtenberg and Stanley S. Wainer. Tiered Arithmetics. In G. Jäger and W. Sieg, editors, *Feferman on Foundations*, volume 13, pages 145–168. Springer International Publishing, 2017.

[26] Helmut Schwichtenberg and Franziskus Wiesnet. Logic for exact real arithmetic. *Logical Methods in Computer Science*, 17(2):7:1–7:27, April 2021.

[27] Dana S. Scott. *Domains for denotational semantics*, pages 577–610. Springer Berlin Heidelberg, 1982.

[28] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1):411–440, 1993.

[29] Josef Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405, February 1967.

[30] Anne Sjerp Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Springer Berlin Heidelberg, 1973.

[31] Franziskus Wiesnet. Konstruktive Analysis mit exakten reellen Zahlen. Master's thesis, Ludwig-Maximilians Universität München, September 2017.

[32] Franziskus Wiesnet. Introduction to Minlog. In Klaus Mainzer, Peter Schuster, and Helmut Schwichtenberg, editors, *Proof and Computation*, pages 233–288. World Scientific, May 2018.

[33] Franziskus Wiesnet. Minlog-Kurs. https://www.youtube.com/playlist?list=PLD87fNDrm1skaFxUA-ArQjmqj50IARRPf, 2024. YouTube Playlist.

[34] Franziskus Wiesnet. Verified Program Extraction in Number Theory: The Fundamental Theorem of Arithmetic and Relatives. *arXiv*, April 2025. https://arxiv.org/abs/2504.03460, (under review).

[35] Franziskus Wiesnet and Nils Köpp. Limits of real numbers in the binary signed digit representation. *Logical Methods in Computer Science*, 18(3), August 2022.

Linear Types inside Dependent Type Theory

Maximilian Doré

Department of Computer Science, University of Oxford, United Kingdom
maximilian.dore@cs.ox.ac.uk

Abstract

We propose a novel approach to combining linear and dependent type theory. By deeply embedding the rules of linear logic inside dependent type theory, we obtain a linear type system which is inherently also dependent. Moreover, we can dynamically compute resource notations, which allows us to give precise types to programs which need a number of copies of some input depending on some other input. We demonstrate our approach with an implementation in Cubical Agda that allows us to program in a practical way. We then propose a novel type theory which also has dependent linear function types.

Background Ever since the rise of dependent type theory and linear logic, the prospect of having a type theory that has both predicates and allows for restricting variable use has inspired research into *dependent linear type theories* [2, 3, 12, 4]. More recently, quantitative [5, 1] and graded type theories [6, 9] have been proposed as practical programming languages in which users can specify in the type of a program how often the program uses a given input, we call this the *multiplicity* of this input. In many cases, the multiplicity of some input depends on the value of some other input, consider for example the following Haskell program.

```
safeHead :: [a] -> a -> (a,[a])
safeHead [] y = (y,[])
safeHead (x:xs) y = (x,xs)
```

The program uses the backup element `y` only in case the given list is empty. However, systems which have static multiplicities such as quantitative and graded type theories [1, 6, 9] do not allow for precisely capturing this in the type system.

We propose a new approach of combining dependent type theory with linear logic that allows for equipping inputs with multiplicities that depend on the values of other inputs. The main idea behind our system is to deeply embed linear logic in dependent type theory and have the structural rules of linear logic apply to terms of the host theory. More precisely, given a context Γ of a standard dependent type theory, we require a symmetric monoidal category Supply_Γ with bifunctor $\otimes$, plus a bit more structure to be made precise below. We call an object Δ of Supply_Γ a *supply*, and its morphisms *productions* where we write $\Delta_0 \rhd \Delta_1$ for the collection of morphisms between Δ_0 and Δ_1. We impose this linear structure in the host dependent type theory, i.e., each Supply_Γ and $(-) \rhd (-)$ are themselves types, and its objects/morphisms are terms. Lastly, any term that is derivable in Γ can be considered a singleton supply, i.e., for any given $\Gamma \vdash a : A$ we have a $\iota(a) : \mathsf{Supply}_\Gamma$.

Using this structure, we define linear entailment as the following dependent type.

$$\Delta \Vdash A \coloneqq \Sigma(a : A)(\Delta \rhd \iota(a))$$

In words, to conclude A from a supply Δ, we need to give a term $a : A$ as well as a production that turns the supply into this term. We can regard our system as having two nested entailments, where $\vdash$ is intuitionistic entailment and $\Vdash$ is linear entailment.

$$\Gamma \vdash \Delta \Vdash A$$

Having the linear judgment as a dependent type in the host theory has two crucial advantages: we can use open terms of the host theory to compute supplies, which allows for dynamic multiplicities; and we can derive linear elimination principles using normal dependent elimination, which simplifies the introduction of data types in our system in contrast to quantitative type theories [7].

We can implement Supply_Γ as the finite multiset of pointed types in Cubical Agda [8, 13], which we will sketch in Section 1. To add function types to our system, we need Supply_Γ to have exponentials and a variable binding principle, which means we cannot define this in Cubical Agda anymore and need to devise a new type theory, which we will outline in Section 2. An experimental implementation of our system is online: `https://github.com/maxdore/dltt`.

The observation that equipping the output of a function with a bag of resources gives rise to dynamic multiplicities is due to Pierre-Marie Pédrot [10, 11]. We show how to implement this idea in Cubical Agda; and build on it to devise a novel dependent linear type theory.

1. Linear Types in Cubical Agda Cubical Agda's higher inductive types allow for defining finite multisets over some type. We define *supplies* as finite multisets of pointed types, which allows us to put any term in a supply.

```
Supply : Type
Supply = FMSet (Σ[ A ∈ Type ] A)
```

We can readily define functions for constructing the supply containing a single term a, written $\iota\ a$, and for joining two supplies Δ_0 and Δ_1, written $\Delta_0 \otimes \Delta_1$. We can compute the supply containing n copies of some supply for a given natural number n with a straightforward recursive definition.

```
_^_ : Supply → ℕ → Supply
Δ ^ zero = ◇
Δ ^ (suc n) = Δ ⊗ (Δ ^ n)
```

Supply can be regarded as a symmetric monoidal category whose laws hold up to propositional equality. However, we will need to add more morphisms between supplies to take into account constructors of data types, which is why we introduce a dedicated type of morphisms, called *productions*. This type will be extended with other constructors, we only give its main constructors here.

```
data _▷_ : Supply → Supply → Type where
  id : ∀ Δ → Δ ▷ Δ
  _∘_ : ∀ {Δ₀ Δ₁ Δ₂} → Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
  _⊗ᶠ_ : ∀ {Δ₀ Δ₁ Δ₂ Δ₃} → Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → Δ₀ ⊗ Δ₂ ▷ Δ₁ ⊗ Δ₃
```

Every supply can be turned into itself with id (which allows us to lift equalities between supplies to productions), while $\circ$ and $\otimes^f$ give transitivity and congruence principles for productions. We have omitted equality rules such as id being the unit for composition, these follow in a standard way for symmetric monoidal categories.

Using this structure, we can define our linear judgment as a dependent type as follows.

```
_⊩_ : Supply → Type → Type
Δ ⊩ A = Σ[ a ∈ A ] (Δ ▷ ι a)
```

We can conveniently program using this notion of linear judgment. For example, we can implement an analogue of `safeHead` from above in Cubical Agda and give it the following type.

$$\mathsf{safeHead} : (xs : \mathsf{List}\ A) \to (y : A) \to \iota\ xs \otimes (\iota\ y)\ \hat{}\ \mathsf{null}\ xs \Vdash A \times \mathsf{List}\ A$$

We need a single instance of xs in our program, while the multiplicity of y depends on whether xs is null, which is a program that returns 1 if the given list is empty and 0 otherwise. To implement safeHead, we need to add more productions to $\_\rhd\_$, e.g., a rule to remove a cons constructor from a supply $\iota\ (x :: xs) \rhd \iota\ x \otimes \iota\ xs$. This rule captures that the free variables of a non-empty list are the same as the free variables of head and tail considered separately.

2. Linear dependent functions In order to add function types to our system, we need additional structure which is not present in Cubical Agda. First, we require that our supplies have exponentials, i.e., each Supply_Γ is a symmetric monoidal closed category where we write $[\Delta_0, \Delta_1]$ for the supply which internalises productions between Δ_0 and Δ_1. Second, we need to be able to bind free variables in supplies, i.e., we require a functor

$$\Lambda_{x:A} : \mathsf{Supply}_{\Gamma,x:A} \to \mathsf{Supply}_\Gamma$$

Furthermore, $\Lambda_{x:A}$ has to be right adjoint to context extension of supplies (context extension of dependent type theory entails that any term of Supply_Γ is also a term of $\mathsf{Supply}_{\Gamma,x:A}$). Using this structure we can define dependent linear function types as follows.

$$\begin{aligned}&(-) \multimap (-) : (A : \mathsf{Type}) \to (B : A \to \mathsf{Type}) \to \Sigma(C : \mathsf{Type})(C \to \mathsf{Supply})\\ &(x : A) \multimap B(x) = ((x : A) \to B(x))\ ,\ (\lambda f \to \Lambda_{x:A}[\iota(x), \iota(f\ x)])\end{aligned} \tag{1}$$

In words, a dependent linear function is a dependent function f and a production that witnesses that any input $x : A$ represents the same resources as the output of applying f to x. To iterate this function type, we need to slightly generalise the above definition, we refer the interested reader to the formalisation.

We can derive natural introduction and elimination principles for our functions.

$$\frac{\Gamma, x : A \vdash \Delta \otimes \iota(x) \Vdash b : B(x)}{\Gamma \vdash \Delta \Vdash \lambda x.b : (x : A) \multimap B(x)}(x \notin \mathsf{fv}(\Delta)) \qquad \frac{\Gamma \vdash \Delta_0 \Vdash f : (x : A) \multimap B(x) \qquad \Gamma \vdash \Delta_1 \Vdash a : A}{\Gamma \vdash \Delta_0 \otimes \Delta_1 \Vdash f\ a : B(a)}$$

These rules can be generalised to take in n copies of the input for some open term n of the natural numbers, we write $(x : A)^n \multimap B$ for such a function. Using these functions with multiplicities, we can write safeHead from above as a proper linear dependent function.

$$\mathsf{safeHead} : (xs : \mathsf{List}\ A)^1 \multimap (y : A)^{\mathsf{null}\ xs} \multimap A \times \mathsf{List}\ A$$

Our system therefore has both dependent types and *dependent multiplicities*, giving an expressive language to type many programs that cannot be precisely typed otherwise.

Acknowledgments I am indebted to Valeria de Paiva and Pierre-Marie Pédrot for introducing me to the Dialectica construction; and to Nathan Corbyn and Daniel Gratzer for clearing up my type-theoretic confusions. I am grateful for helpful discussions with Thorsten Altenkirch, Pedro H. Azevedo de Amorim, Evan Cavallo, Jeremy Gibbons, Sean Moss and Sam Staton; and for the comments of several anonymous reviewers.

References

[1] Robert Atkey. Syntax and semantics of quantitative type theory. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, (LICS):56–65, 2018.

[2] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and computation*, 179(1):19–75, 2002.

[3] Neelakantan R Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. *Principles of Programming Languages 2015 (POPL)*, 50(1):17–30, 2015.

[4] Daniel R. Licata, Michael Shulman, and Mitchell Riley. A Fibrational Framework for Substructural and Modal Logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:22, Dagstuhl, Germany, 2017.

[5] Conor McBride. I got plenty o'nuttin'. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 207–233, 2016.

[6] Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded modal dependent type theory. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 462–490, Cham, 2021. Springer International Publishing.

[7] Georgi Nakov and Fredrik Nordvall Forsberg. Quantitative polynomial functors. In Henning Basold, Jesper Cockx, and Silvia Ghilezan, editors, *27th International Conference on Types for Proofs and Programs (TYPES)*, volume 239 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, Dagstuhl, Germany, 2022.

[8] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

[9] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–30, 2019.

[10] Pierre-Marie Pédrot. A functional functional interpretation. *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science*, (LICS/CSL), 2014.

[11] Pierre-Marie Pédrot. Dialectica the ultimate. Talk at Trends in Linear Logic and Applications (https://www.p%C3%A9drot.fr/slides/tlla-07-24.pdf), 2024.

[12] Matthijs Vákár. A categorical semantics for linear logical frameworks. In Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, pages 102–116, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[13] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31:e8, 2021.

Impredicative Encodings of (Co)inductive Types

Steven Bronsveld[1], Herman Geuvers[12], and Niels van der Weide[1]

[1] iCIS, Radboud University Nijmegen, The Netherlands
[2] Technical University Eindhoven, The Netherlands

Abstract

In impredicative type theory (System F, also known as $\lambda 2$, [3, 4]), it is possible to define inductive data types, such as natural numbers. It is also possible to define coinductive data types such as streams. Their (co)recursion principles obey the expected computation rules (the β-rules), but unfortunately, they do not yield a (co)induction principle [2, 6], because the necessary uniqueness principles are missing (the η-rules). Awodey, Frey, and Speight [1] used a extension of the Calculus of Constructions with Σ-types, identity-types, and functional extensionality to define System F style inductive types with an induction principle, by encoding them as a well-chosen subtype, making them initial algebras. We extend their results to coinductive data types, and below we detail the stream data type with the desired coinduction principle (also called bisimulation). To do that, we first define quotient types (with the desired η-rules) and we also need a stronger form of the definable existential types. The method of [1] can be used for general inductive types by defining W-types with an induction principle. The dual approach for streams can be extended to M-types, the generic notion of coinductive types, and the dual of W-types.

To define final coalgebras impredicatively, we first need quotient types. The well-known impredicative definition of quotient types satisfies the β-rule, but fails to satisfy the η-rule. We extend the technique of [1] to define quotients with an η-rule. We fix a type $D : \mathcal{U}$ and a relation $R : D \to D \to \mathcal{U}$. We say that a function f **respects** R (as a type: $\mathtt{EqCls}\ f\ R$) if for all $x, y : D$ such that $R\ x\ y$, we have $f\ x = f\ y$. The well-known impredicative **quotient type**, $\mathtt{quot}^*\ D\ R$, is defined as $\Pi(C : \mathcal{U}).\Pi(f : D \to C).\mathtt{EqCls}\ f\ R \to C$. The **class function** $\mathtt{cls}^* : D \to \mathtt{quot}^*\ D\ R$ is defined to be

$$\mathtt{cls}^* := \lambda(d : D).\lambda(C : \mathcal{U}).\lambda(f : D \to C).\lambda(H : \mathtt{EqCls}\ f\ R).f\ d.$$

We can lift a function $(f : D \to E)$ that respects R to a function $(\overline{f} : \mathtt{quot}^*\ D\ R \to E)$. This lifting is done by the **recursor for quotient types**, $\mathtt{rec}^*_\mathtt{q}$:

$$\mathtt{rec}^*_\mathtt{q} := \lambda(C : \mathcal{U}).\lambda(f : D \to C).\lambda(H : \mathtt{EqCls}\ f\ R).\lambda(q : \mathtt{quot}^*\ D\ R).q\ C\ f\ H$$

As usual we write $\overline{f}$ for $(\lambda q.\ \mathtt{rec}^*_\mathtt{q}\ C\ f\ H\ q)$. We have that $\overline{f} \circ \mathtt{cls}^* = f$, which gives the β-rule for quotients. NB. We do not require R to be an equivalence relation, so $\mathtt{quot}^* DR$ is actually the quotient of D by the smallest equivalence relation containing R.

The **inductive quotient type** (that will satisfy the η-rule), which we denote by $\mathtt{quot}$, is defined as $\Sigma(q : \mathtt{quot}^*\ D\ R).\mathtt{LimQuot}\ q$ where $\mathtt{LimQuot}\ (q : \mathtt{quot}^*\ D\ R)$ says that for all functions $g : D \to X$ and $g' : D \to Y$ that respect R, and functions $f : X \to Y$, we have $f\ (\mathtt{rec}^*_\mathtt{q}\ X\ g\ H\ q) = \mathtt{rec}^*_\mathtt{q}\ Y\ g'\ H'\ q$ whenever $f \circ g = g'$. We define the **recursor** as follows.

$$\mathtt{rec}_\mathtt{q} := \lambda(C : U).\lambda(f : D \to C).\lambda(H : \mathtt{EqCls}\ f\ R).\lambda(q : \mathtt{quot}\ D\ R).\mathtt{rec}^*_\mathtt{q}\ C\ f\ H\ (\mathtt{pr1}\ q)$$

It can be shown that we have a term $\mathtt{LimQuotCls} : (\mathtt{LimQuot}\ (\mathtt{cls}\ d))$, and using that we define the **class function** $\mathtt{cls}$ to be $\lambda(d : D).\langle \mathtt{cls}^*\ d,\ \mathtt{LimQuotCls}\ d\rangle$. We can now show that the general η-rule is satisfied: If $(g : D \to C)$ respects R, then for every $(f : D/R \to C)$ with $f \circ \mathtt{cls} = g$, we have $f = \overline{g}$.

It is standard to define the **impredicative existential type** $\exists^*(X:\mathcal{U}).P(X)$ to be $\Pi(Y:\mathcal{U}).(\Pi(X:\mathcal{U}).(P\,X)\to Y)\to Y$. The constructor $\mathtt{pack}^*$ and recursor $\mathtt{rec}^*_\exists$ are defined to be

$$\begin{array}{rcl}\mathtt{pack}^* & := & \lambda(X:\mathcal{U}).\lambda(t:P\,X).\lambda(Y:\mathcal{U}).\lambda(k:\Pi(X:\mathcal{U}).(P\,X)\to Y).k\ X\ t\\ \mathtt{rec}^*_\exists & := & \lambda(Y:\mathcal{U}).\lambda(f:\Pi(Z:\mathcal{U}).(P\,Z)\to Y).\lambda(e:\exists^*X.P).e\ Y\ f\end{array}$$

These do not satisfy the desired equations: when we *unpack* an $(e:\exists X.P)$ using the recursor $\mathtt{rec}_\exists$, obtaining $(X:\mathcal{U})$ and $(t:P(X))$, and then we re-package them into $e' := \mathtt{pack}\ X\ t$, we want $e = e'$. We can define an improved existential type that satisfies the properties that we need: For all $e:\exists X.P$ there are $X:\mathcal{U}$ and $t:P\,X$ such that $e = \mathtt{pack}\,X\,t$. In addition, $\mathtt{rec}_\exists\ (\exists X.P)\ \mathtt{pack} = \mathtt{id}_{\exists X.P}$.

The well-known **impredicative stream type** $\mathtt{Stream}^*$ (over a carrier type E) is defined as $\exists(X:\mathcal{U}).X\times(X\to E)\times(X\to X)$. The destructors $\mathtt{hd}^*:\mathtt{Stream}^*\to E)$ and $\mathtt{tl}^*:\mathtt{Stream}^*\to\mathtt{Stream}^*$ are standard to define, and so is the **corecursor** (that produces a stream):

$$\mathtt{corec}^*_{\mathtt{s}} := \lambda(X:\mathcal{U}).\lambda(h:X\to E).\lambda(t:X\to X).\lambda(x:X).\mathtt{pack}\ X\ \langle x,h,t\rangle.$$

It is a simple check that the introduction rule (corecursor) and elemination rules (head and tail functions) compute as one would expect. We want to define a stream-type that satisfies the η-rule, which is taken from the final coalgebra: Given a stream-morphism $(f:X\to Y)$, we want that $u_Y\circ f = u_X$ for the morphisms $u_X := \mathtt{corec}^*_{\mathtt{s}}\ X\ h\ t$ and $u_Y := \mathtt{corec}^*_{\mathtt{s}}\ Y\ h'\ t'$.

In the case of quotients (and for natural numbers, see [1]), we create a *subtype* of $\mathtt{quot}^*\ D\ R$. In the case of streams, we take the dual notion: a *quotient*. To define this quotient, we define a relation $\mathtt{CoLimStr}$ that relates streams σ and τ if they can be translated into each other by some stream-morphism.

$$\begin{array}{rcl}\mathtt{CoLimStr}\ \sigma\ \tau & := & \exists(X,Y:\mathcal{U}).\exists^{(h:X\to E)}_{(h':Y\to E)}.\exists^{(t:X\to X)}_{(t':Y\to Y)}.\exists(f:X\to Y).\exists(x:X)\\ & & (\mathtt{MorphStream}\ X\ h\ t\quad Y\ h'\ t'\quad f)\ \wedge\\ & & \sigma = \mathtt{corec}^*_{\mathtt{s}}\ X\ h\ t\ x\ \wedge\ \tau = \mathtt{corec}^*_{\mathtt{s}}\ Y\ h'\ t'\ (f\ x)\end{array}$$

We use the infix notation $(\sigma\ \equiv\ \tau)$ to denote that $(\mathtt{CoLimStr}\ \sigma\ \tau)$ holds. The relation $\mathtt{CoLimStr}$ is neither *symmetric* nor *transitive*. This does not hinder us since the equality relation on the quotient is an equivalence relation and we will have $\mathtt{cls}\,\sigma = \mathtt{cls}\,\tau$ if $(\mathtt{CoLimStr}\ \sigma\ \tau)$ holds. The **coinductive stream type** $\mathtt{Stream}$ is defined to be $\mathtt{quot}\ \mathtt{Stream}^*\ \mathtt{CoLimStr}$. To define the new head and tail functions as lifted functions of $\mathtt{hd}^*$ and $\mathtt{tl}^*$ we need to show that $\mathtt{hd}^*$ and $\mathtt{tl}^*$ respect $\equiv$. Using that we define the destructors for the new stream type: $\mathtt{hd}:\mathtt{Stream}\to E$ and $\mathtt{tl}:\mathtt{Stream}\to\mathtt{Stream}$. Finally, we define the corecursor $\mathtt{corec}_{\mathtt{s}}:\Pi(X:\mathcal{U}).(X\to E)\to(X\to X)$ to be $\mathtt{cls}\circ\mathtt{corec}^*_{\mathtt{s}}$. Using this we can show the η-rule for $\mathtt{Stream}^*$: For all $(X:\mathcal{U}),(h:X\to E),(t:X\to X)$ and $(f:X\to\mathtt{Stream})$, if $(\mathtt{MorphStream}\ X\ h\ t\ \mathtt{Stream}\ \mathtt{hd}\ \mathtt{tl}\ f)$, then we have $(f = \mathtt{corec}_{\mathtt{s}}\ X\ h\ t)$.

Two streams σ,τ are bisimilar, $\sigma\sim\tau$, if there is a bisimulation relation that relates them, where R is a **bisimilation** if for all $\sigma,\tau:\mathtt{Stream}$ we have that $\mathtt{hd}\ \sigma = \mathtt{hd}\ \tau$ and $R\ (\mathtt{tl}\ \sigma)\ (\mathtt{tl}\ \tau)$ whenever $R\ \sigma\ \tau$. We have the *coinduction* proof principle, stating that bisimilarity and equality coincide. More precisely: for all $\sigma,\tau:\mathtt{Stream}$, we have $\sigma\sim\tau$ if and only if $\sigma=\tau$.

We used the limit predicate to encode the inductive data types, similar to [1]. For coinductive data types, we dually used the colimit. It is also possible to directly encode the induction principle within the embedding, by creating a predicate $\mathtt{Ind}$ as was done in [5]. Similarly for coinductive types, one can use the $\mathtt{CoInd}$ principle by quotienting with bisimilarity relation. It is also possible to encode the η-rule directly by representing the uniqueness requirement, using a predicate $\mathtt{Unq}$.

References

[1] Steve Awodey, Jonas Frey, and Sam Speight. Impredicative encodings of (higher) inductive types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 76–85. ACM, 2018.

[2] Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2001.

[3] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1993.

[4] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[5] Xavier Ripoll Echeveste. Alternative impredicative encodings of inductive types. Master's thesis, Universiteit van Amsterdam, 2023.

[6] Ivar Rummelhoff. Polynat in PER models. *Theor. Comput. Sci.*, 316(1):215–224, 2004.

NbE for LNL via Adjoint Meta-modalities

James Wood[1]

Huawei Research Centre, Edinburgh, UK
lamudri@gmail.com

Abstract

Linear/non-Linear Logic (LNL) was introduced by Benton [1994], and is a calculus comprising both linear and intuitionistic terms. In this work, we give an intrinsically typed Agda-mechanised (see Wood [2025] for the code) normalisation by evaluation (NbE) [Berger and Schwichtenberg, 1991] procedure for LNL, using metalanguage constructs F and G to mediate between linear and intuitionistic parts throughout the metatheory.

Overview. We give an NbE procedure based on that for Simply Typed λ-Calculus given by Allais et al. [2017], in particular reusing the context-implicit notational style of that work. Our model construction and *reify*, *reflect*, and *evaluation* functions appear to be very similar, just with refinements to deal with linearity and modifications for the choice of object language connectives. However, the underlying definitions and lemmas, particularly pertaining to *environments*, are different and new, and are our main focus in this extended abstract.

Contexts. We take the LNL types of Benton [1994], split into *linear* and *intuitionistic* types. We let intuitionistic contexts (Θ and Λ) be lists of intuitionistic types, and let linear contexts (Γ and Δ) be lists of arbitrary types tagged **lin** or **int** as appropriate. The two kinds of context can be related by $\sim$, with $\Gamma \sim \Theta$ whenever all types in Γ are intuitionistic and Γ and Θ are pointwise equal. Note that $\sim$ is functional in both directions and total from right to left.

We often consider families of sets indexed over contexts, with $\mathcal{L}$Fam and $\mathcal{I}$Fam being the sets indexed over linear and intuitionistic contexts respectively. We have operators on $\mathcal{L}$Fam and $\mathcal{I}$Fam called *meta-connectives*, coloured blue. These include the *proof relevant separation logic* connectives of Rouvoet et al. [2020] — namely *separating conjunction* $*$, *separating unit* I, and *separating implication* $-\!\!*$ — on $\mathcal{L}$Fam. Acting on both $\mathcal{L}$Fam and $\mathcal{I}$Fam are the pointwise operators $\dot{\times}$, $\dot{\to}$, and $\dot{\cup}$. We also have the *meta-modalities* F and G of definition 1. Binders Σ and Π stand for the dependent pair/function type formers of the ambient theory (Agda).

Definition 1 (Meta-modalities)**.** *Let* $\mathsf{F} : \mathcal{I}\mathrm{Fam} \to \mathcal{L}\mathrm{Fam}$ *be defined by* $(\mathsf{F}\,T)\,\Gamma \coloneqq \Sigma\Theta.\ \Gamma \sim \Theta \times T\,\Theta$*, and let* $\mathsf{G} : \mathcal{L}\mathrm{Fam} \to \mathcal{I}\mathrm{Fam}$ *be defined by* $(\mathsf{G}\,T)\,\Theta \coloneqq \Sigma\Gamma.\ \Gamma \sim \Theta \times T\,\Theta$*.*

Our first use of the meta-modalities is in defining the object-language (LNL) modalities: F and G. The introduction forms use the corresponding meta-modalities directly, similarly to how $\otimes$-introduction uses $*$ directly to split the context between the two subterms. G-elimination also uses meta-F directly, to restrict the rule's use to purely intuitionistic contexts. However, F-elimination does not use any meta-modalities, instead using binding of an intuitionistic variable to cross modes. F-elimination does, nevertheless, use meta-connective $*$ to split the free linear variables between the two premises. Each of these rules is understood to take an arbitrary ambient linear/intuitionistic (as appropriate) context, which is the context of the conclusion. The contexts of the premises are derived from the ambient context via the meta-connectives.

$$\frac{\mathsf{F}\,(\vdash_{\mathcal{I}} X)}{\vdash_{\mathcal{L}} \mathsf{F}X}\ \mathrm{FI} \qquad \frac{\mathsf{G}\,(\vdash_{\mathcal{L}} A)}{\vdash_{\mathcal{I}} \mathsf{G}A}\ \mathrm{GI} \qquad \frac{\vdash_{\mathcal{L}} \mathsf{F}X \ * \ \mathbf{int}\,X \vdash_{\mathcal{L}} A}{\vdash_{\mathcal{L}} A}\ \mathrm{FE} \qquad \frac{\mathsf{F}\,(\vdash_{\mathcal{I}} \mathsf{G}A)}{\vdash_{\mathcal{L}} A}\ \mathrm{GE}$$

Aside from modalities F and G, we have linear and intuitionistic function types, negative intuitionistic products, and positive linear tensor products, all with rules equivalent to those of Benton [1994, Fig. 3]. We also have linear and intuitionistic base types, with no rules.

Variables and environments. The variable judgements are defined as follows. The variable judgements embed into their respective term judgements via the variable rules (not pictured).

Definition 2 (Variables)**.** *Let* $\Theta \ni_{\mathcal{I}} X$ *be the set of entries of* Θ *equal to* X*. Let* $\Gamma \ni_{\mathcal{L}} A$ *be a singleton if there is exactly one linear entry in* Γ *and that entry has type* A*, and empty otherwise.*

We choose very intensionally different representations of environments in the intuitionistic and linear cases. This is partially to avoid any coincidences and sleights of hand when passing between environment types, but also reflects the path of least resistance in each mode considered separately. The functional definition given in definition 3 is standard, but unavailable in the linear case without some overarching linearity condition [Wood, 2024, sec. 5.1]. These definitions are parametrised on an intuitionistic judgement form $\triangleright$ and a linear judgement form $\blacktriangleright$. Both of these judgement forms are indexed on a context and a type, like the variable and term judgement forms, where the context and type are intuitionistic for $\triangleright$ and linear for $\blacktriangleright$.

Definition 3 (Intuitionistic environment)**.** *Let* $\Theta \overset{\triangleright}{\Longrightarrow}_{\mathcal{I}} \Lambda \coloneqq \Pi X.\ \Lambda \ni_{\mathcal{I}} X \to \Theta \triangleright X$*.*

Definition 4 (Linear environment)**.** *Let* $(-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta$ *be defined inductively by the following inclusions, where* $\forall[\ T\ ] \coloneqq \Pi\Gamma.\ T\,\Gamma$ *and* $(S \dot{\to} T)\ \Gamma \coloneqq S\,\Gamma \to T\,\Gamma$*:*

$$\forall\Big[\ \mathrm{I} \dot{\to} (-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \cdot\ \Big] \qquad \forall\Big[\ (-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta * \quad (-) \blacktriangleright A \quad \dot{\to} \quad (-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta, \mathbf{lin}\, A\ \Big]$$

$$\forall\Big[\ (-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta * \mathrm{F}\,((-) \triangleright X) \quad \dot{\to} \quad (-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta, \mathbf{int}\, X\ \Big]$$

We convert between linear and intuitionistic contexts using the following lemma. We only convert at purely intuitionistic contexts, as enforced by the meta-modalities, so the linear value judgement $\blacktriangleright$ does not matter. Note that it is somewhat unusual to apply meta-connectives to a family with its open place on the *right* of the environment judgements.

Lemma 5. *Given linear contexts* Γ, Δ *and intuitionistic contexts* Θ, Λ*, we have a functions from* $\mathrm{G}\left(\Gamma \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} (-)\right)\ \Lambda$ *to* $\mathrm{F}\left((-) \overset{\triangleright}{\Longrightarrow}_{\mathcal{I}} \Lambda\right)\ \Gamma$ *and from* $\mathrm{F}\left(\Theta \overset{\triangleright}{\Longrightarrow}_{\mathcal{I}} (-)\right)\ \Delta$ *to* $\mathrm{G}\left((-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta\right)\ \Theta$*.*

On the intuitionistic side, distribution of an environment between subterms is trivial (by copying the whole environment). On the linear side, however, we have that an environment into a context which splits yields a splitting of the source context and two smaller environments.

Lemma 6. *If we have an environment of type* $\Gamma \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta$ *and* Δ *splits into* Δ_l *and* Δ_r*, then we have* $\left((-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta_l * (-) \overset{\blacktriangleright,\triangleright}{\Longrightarrow}_{\mathcal{L}} \Delta_r\right)\ \Gamma$*. If, instead of splitting,* Δ *contains no linear assumptions, then* Γ *also contains no linear assumptions, i.e.,* $\mathrm{I}\,\Gamma$*.*

We use environments to define both renamings (environments in which the values are variables) and evaluation environments (environments in which the values are elements from the NbE model). With renamings come the meta-modalities $\Box_{\mathcal{L}}$ and $\Box_{\mathcal{I}}$, which take a family and produce the largest family stable under renaming contained in it, as per Allais et al. [2017].

NbE. Our NbE model is given by families $\vDash_{\mathcal{L}}$ and $\vDash_{\mathcal{I}}$, defined below with the contexts left implicit and manipulated via meta-connectives. Once again, meta-F and meta-G help in interpreting object-F and object-G, and similar relationships hold for other meta-connectives and types. As is standard, both function types have their interpretation coerced into being renameable, and positive types may be interpreted as neutral terms as well as semantic values.

$$
\begin{array}{llll}
\vDash_{\mathcal{L}} \iota_{\mathcal{L}} & := \vdash^{\mathrm{ne}}_{\mathcal{L}} \iota_{\mathcal{L}} & \vDash_{\mathcal{I}} \iota_{\mathcal{I}} & := \vdash^{\mathrm{ne}}_{\mathcal{I}} \iota_{\mathcal{I}} \\
\vDash_{\mathcal{L}} A \otimes B & := (\vDash_{\mathcal{L}} A * \vDash_{\mathcal{L}} B) \mathbin{\dot\cup} \vdash^{\mathrm{ne}}_{\mathcal{L}} A \otimes B & \vDash_{\mathcal{I}} X \times Y & := \vDash_{\mathcal{I}} X \mathbin{\dot\times} \vDash_{\mathcal{I}} Y \\
\vDash_{\mathcal{L}} A \multimap B & := \Box_{\mathcal{L}} (\vDash_{\mathcal{L}} A \mathbin{-\!\!*} \vDash_{\mathcal{L}} B) & \vDash_{\mathcal{I}} X \to Y & := \Box_{\mathcal{I}} (\vDash_{\mathcal{I}} X \mathbin{\dot\to} \vDash_{\mathcal{I}} Y) \\
\vDash_{\mathcal{L}} \mathrm{F} X & := \mathrm{F} (\vDash_{\mathcal{I}} X) \mathbin{\dot\cup} \vdash^{\mathrm{ne}}_{\mathcal{L}} \mathrm{F} X & \vDash_{\mathcal{I}} \mathrm{G} A & := \mathrm{G} (\vDash_{\mathcal{L}} A)
\end{array}
$$

With these definitions, the *reify* and *reflect* functions follow largely as may be expected. The evaluator $\mathit{eval}_{\mathcal{L}} : \Gamma \overset{\vDash_{\mathcal{L}}, \vDash_{\mathcal{I}}}{\Longrightarrow}_{\mathcal{L}} \Delta \to \Delta \vdash_{\mathcal{L}} A \to \Gamma \vDash_{\mathcal{L}} A$ (and similar for the intuitionistic mode) is somewhat more involved, and uses most of the properties and operations established about meta-connectives (including their functoriality), environments, and renaming. Details can be found in the associated artefact [Wood, 2025].

References

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, page 195–207, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450347051. doi: 10.1145/3018610.3018613. URL https://doi.org/10.1145/3018610.3018613.

P.N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. pages 121–135. Springer-Verlag, 1994.

Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 203–211. IEEE Computer Society, 1991. doi: 10.1109/LICS.1991.151645. URL https://doi.org/10.1109/LICS.1991.151645.

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *CPP 2020*, pages 284–298, 2020. ISBN 9781450370974. doi: 10.1145/3372885.3373818.

James Wood. *A framework for semiring-annotated type systems*. PhD thesis, 2024. URL https://doi.org/10.48730/pa0y-ct71.

James Wood. NbE for LNL via adjoint meta-modalities (artefact), 2025. URL https://github.com/laMudri/lin-env/blob/types25-submission/src/Modal/LnL.agda.

Unboxed Dependent Types

Constantine Theocharis and Ellis Kesterton

University of St Andrews
kt81@st-andrews.ac.uk

When the reduction rules of type theory become part of its static semantics, it is sometimes impossible to compute the memory size in bytes of a type during compilation. This happens when polymorphism is present but monomorphisation is not, and in particular when dependent types are part of the mix, since the dependency might be on runtime values.

In this ongoing work, we formulate a dependent type theory where the memory size of a type is always known at compile-time, and thus compilation can directly target a low-level language. Boxing is opt-in and can be avoided, leading to efficient code that enables cache locality. This is done by *indexing* syntactic types by a metatheoretic type describing memory layouts: Bytes. This leads to a notion of representation polymorphism. Unboxed data in functional languages has been explored before [7, 6, 5], but results in overly restricted polymorphism or complex theories with multiple levels of kinds separating values and computations, and without support for full dependent types. Our approach is lightweight and extends to dependent types.

The staging view of *two-level type theory* (2LTT) [3] has been explored by Kovács in the general setting [9] as well as in the setting of closure-free functional programming [10]. Inspired by a note in the aforementioned works, we can embed our unboxed type theory as the object language of a 2LTT, which allows us to write type-safe metaprograms that compute representation-specific constructions. For example, we can formulate a universe of flat protocol specifications in the style of Allais [2], and interpret it in the unboxed object theory. We needn't compromise on the usage of dependent types either; as opposed to [10], the object theory is itself dependently typed and thus we can encode unboxed higher-order polymorphic functions as part of the final program, because all *sizes* (not necessarily all types) are known after staging.

Basic setup We formulate our system in the form of a second-order generalised algebraic theory (SOGAT) [11]. We assume a metatheoretic type of Bytes with

$$0, 1 : \mathsf{Bytes} \qquad + : \mathsf{Bytes} \to \mathsf{Bytes} \to \mathsf{Bytes} \qquad \mathsf{ptr} : \mathsf{Bytes} \qquad \times : \mathsf{Nat} \to \mathsf{Bytes} \to \mathsf{Bytes}\,.$$

The constant ptr defines the size of a pointer. Any model of the signature above will suffice; such a model might encode a sophisticated layout algorithm with padding, for example.

To begin with, types are indexed by bytes:[1] $\mathsf{Ty} : \mathsf{Bytes} \to \mathbf{Set}$. We have the following basic type and term formers:

$$\mathsf{U}_{\_} : \mathsf{Bytes} \to \mathsf{Ty}\ 0 \qquad \mathsf{Tm}\ \mathsf{U}_b = \mathsf{Ty}\ b$$
$$\Box : \mathsf{Ty}\ b \to \mathsf{Ty}\ \mathsf{ptr} \qquad \mathsf{box} : \mathsf{Tm}\ \Box A \simeq \mathsf{Tm}\ A : \mathsf{unbox}\,.$$

The universe U_b describes types whose inhabitants take up b bytes, with a Grothendiek-style identification $\mathsf{Tm}\ \mathsf{U}_b = \mathsf{Ty}\ b$. The $\Box$ type former takes sized types to types which *box* their contents. In other words, for a type A of some size a, an inhabitant of data of $\Box A$ will be stored on the heap behind a pointer indirection. On the other hand, an inhabitant of A will be stored *inline* on the stack, since it is known that A takes up exactly a bytes. Codes for types of any kind take up no space at runtime because they are erased. Additionally, we have

[1] For brevity we will not regard issues of universe sizing, but this can be accomodated without issue.

boxing and unboxing operators for types of a known size. Since we present this as a second-order theory, we can mechanically derive its first-order presentation [8], with explicit contexts. There, contexts Γ record the size of each type, such that $\mathrm{b}(\Gamma) : \mathsf{Bytes}$ can be defined as the sum of the sizes of its types. The action of substitutions on types does not vary their sizes: $-[-] : \mathsf{Ty}\ \Gamma\ b \to \mathsf{Sub}\ \Delta\ \Gamma \to \mathsf{Ty}\ \Delta\ b$.

Functions and unboxed pairs This setup can be augmented with Π and Σ types, where the dependency is *uniform* with respect to layout:

$$\Pi : (A : \mathsf{Ty}\ a) \to (\mathsf{Tm}\ A \to \mathsf{Ty}\ b) \to \mathsf{Ty}\ \mathsf{ptr}$$
$$\Sigma : (A : \mathsf{Ty}\ a) \to (\mathsf{Tm}\ A \to \mathsf{Ty}\ b) \to \mathsf{Ty}\ (a + b)\,.$$

For functions, we store a pointer to an allocation containing both code and data. Alternatively, we could separate the two by sizing functions as $\mathsf{ptr} + \mathsf{ptr}$. Conversely, pairs are stored inline; their size is the sum of the sizes of their components. The term formers remain unchanged.

It could be desirable to have a function type with *unboxed* captures, which might look like

$$\Pi_k : (A : \mathsf{Ty}\ a) \to (\mathsf{Tm}\ A \to \mathsf{Ty}\ b) \to (\mathsf{ptr} + k)\,,$$

annotated with the size of its captures k. This is not expressible as a second-order construct because its term former must know about captured variables: $\lambda : (\rho : \mathsf{Sub}\ \Gamma\ \Delta) \to \mathsf{Tm}\ (\Delta \triangleright A)\ B \to \mathsf{Tm}\ \Gamma\ (\Pi_{\mathrm{b}(\Delta)}\ A\ B)[\rho]$. This also means it is not immediately compatible with 2LTT. We leave this as future work, which would likely involve a modality for closed object terms.

First-class byte values with staging This type theory can be embedded as an object language $\mathbb{O}$ of a 2LTT. On the meta level, we have a type former $\mathbb{B} : \mathsf{Ty}_1$ of byte values, and term formers that mirror Bytes. In an empty context, in the first-order formulation, we get an evaluation function $\mathsf{ev} : \mathsf{Tm}_1\ \bullet\ \mathbb{B} \to \mathsf{Bytes}$. Adding Π types to the meta language allows abstraction over byte values. Moreover, the meta level has a type former $\Uparrow_b$: $\mathsf{Ty}_0\ b \to \mathsf{Ty}_1$ for embedding $(B : \mathsf{Tm}_1\ \mathbb{B})$-sized types from the object fragment. If the final program is of the form $p : \mathsf{Tm}_1\ \bullet\ (\Uparrow_k A)$, after staging we get an object term of size $\mathsf{ev}\ k$.

Example: Maybe as a tagged union Let's take a look at how to define the Maybe type internally in such a way that its data is stored contiguously as a tagged union without indirections.[2]. We assume access to a type $\mathsf{Pad}\ b : \mathsf{U}_b$ which is the unit type that takes up b bytes, with sole constructor pad akin to tt, and $\mathsf{Bool} : \mathsf{U}_1$ which takes up a single byte:

$$\mathsf{Maybe} : (T : \mathsf{U}_b) \to \mathsf{U}_{1+b}$$
$$\mathsf{Maybe}\ T = (x : \mathsf{Bool}) \times \mathsf{if}\ x\ \mathsf{then}\ T\ \mathsf{else}\ \mathsf{Pad}\ b$$

$$\mathsf{nothing} : \{T : \mathsf{U}_b\} \to \mathsf{Maybe}\ T \qquad \mathsf{just} : \{T : \mathsf{U}_b\} \to T \to \mathsf{Maybe}\ T$$
$$\mathsf{nothing} = (\mathsf{false}, \mathsf{pad}) \qquad \mathsf{just} = \lambda\ t.\ (\mathsf{true}, t)$$

Computational irrelevance and runtime-sized data Annotating object-level types with bytes provides a convenient way to handle computational irrelevance without further modifying the structure of contexts. This is possible through a monadic modality

$$|-| : \mathsf{Ty}\ b \to \mathsf{Ty}\ 0 \qquad \|-\| : \mathsf{Tm}\ A \to \mathsf{Tm}\ |A|$$

[2] This is similar to the approach of languages such as Rust [1].

which is idempotent by $(A : \mathsf{Ty}\ 0) \to |A| = A$, extending to all zero-sized types. It also has an appropriate eliminator form. With this we can reproduce, for example, quantitative type theory instantiated with the $\{0, \omega\}$ semiring [4]. This means that we can now use object-level types which are entirely erased: $\mathsf{reverse} : \{n : |\mathsf{Nat}|\} \to \mathsf{Vect}\ T\ n \to \mathsf{Vect}\ T\ n$.

Additionally, we often want to handle data whose size is *not* known at compile-time, but is known at runtime; most commonly, heap-backed arrays, but also other dynamically-sized flat data structures. This is achievable by indexing the universe U by partially-static [12] byte values. Object-level types $\mathsf{Ty}\ b$ are now identified only with $\mathsf{U}_{\mathsf{sta}\ b}$ where $\mathsf{sta} : \mathsf{Tm}_1\ \mathbb{B} \to \mathsf{Tm}_1\ \mathbb{B}^{\mathsf{PS}}$. We can then add appropriate type formers to the theory for the construction of runtime-sized data such as pairs. Their inhabitants cannot directly be stored on the stack, but they can be constructed and manipulated on the heap. To do this, boxing is relaxed to allow runtime-sized data, and we must have a type for 'generating' unsized data. We plan on presenting examples of this in our presentation.

Formalisation, semantics and implementation We have formalised most parts of the sketched system by a shallow embedding in Agda, including the irrelevance modality. We have also formulated a semantics in terms of an untyped lambda calculus which is nevertheless *sized* just like the system we presented. Crucially, the sizes of all constructs in this target language must be *non-zero*, which forces us to translate away all zero-sized types. This justifies the irrelevance modality as well as the erased codes for types. We are currently working on a proof-of-concept implementation that targets LLVM.

Acknowledgements We thank the anonymous reviewers for their helpful comments on the presentation of the system and the interaction of Π_k and 2LTT.

References

[1] std::option - Rust. https://doc.rust-lang.org/std/option/. Accessed: 2025-3-7.

[2] Guillaume Allais. Seamless, correct, and generic programming over serialised data. *arXiv [cs.PL]*, 20 October 2023.

[3] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *Math. Struct. Comput. Sci.*, 33(8):688–743, September 2023.

[4] Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 56–65. Association for Computing Machinery.

[5] Paul Downen. Call-by-unboxed-value. *Proc. ACM Program. Lang.*, 8(ICFP):845–879, 21 August 2024.

[6] Richard A Eisenberg and Simon Peyton Jones. Levity polymorphism. *SIGPLAN Not.*, 52:525–539.

[7] Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, Lecture notes in computer science, pages 636–666. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[8] Ambrus Kaposi and Szumi Xie. Second-order generalised algebraic theories: Signatures and first-order semantics.

[9] András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6(ICFP):540–569, 29 August 2022.

[10] András Kovács. Closure-free functional programming in a two-level type theory. *Proc. ACM Program. Lang.*, 8(ICFP):659–692, 15 August 2024.

[11] Taichi Uemura. Abstract and concrete type theories.

[12] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. Partially-static data as free extension of algebras. *Proc. ACM Program. Lang.*, 2(ICFP):1–30, 30 July 2018.

A Quantitative Dependent Type Theory with Recursion

Oskar Eriksson, Nils Anders Danielsson, Andreas Abel

Chalmers University of Technology and University of Gothenburg, Sweden

In graded modal type theories the type system is parameterized by an algebraic structure, typically some form of semiring, containing *grades*. One use of grades, which we focus on, is to encode quantitative aspects like erasure, linear types, and affine types [3, 4, 5, 6].

In previous work, we have developed an Agda formalization of a graded modal type theory with dependent types [1] supporting Π-types, strong and weak Σ-types, an empty type, natural numbers, and a universe. Certain parts of the syntax, notably lambda abstractions and applications, are annotated with grades corresponding to how many times some part of the program, in this case the function argument, is used. This is checked primarily by a form of typing for grades, the *usage relation* $\gamma \blacktriangleright t$ between a grading context γ and an expression t. Here is a selection of rules (note that our formalization uses de Bruijn indices instead of variables):

$$\frac{}{\mathbf{0}.\, x : 1.\, \mathbf{0} \blacktriangleright x} \qquad \frac{\gamma.\, x : p \blacktriangleright t}{\gamma \blacktriangleright \lambda^p x.t} \qquad \frac{\gamma \blacktriangleright t \qquad \delta \blacktriangleright u}{\gamma + p\delta \blacktriangleright t^p u} \qquad \frac{}{\mathbf{0} \blacktriangleright \mathsf{zero}} \qquad \frac{\gamma \blacktriangleright t}{\gamma \blacktriangleright \mathsf{suc}\, t} \qquad \frac{\gamma \blacktriangleright t}{\delta \blacktriangleright t}\ \delta \leq \gamma$$

In essence, the relation counts the uses of the free variables of t in terms of the operators of the semiring, lifted pointwise to contexts γ. Usage counting is not necessarily exact: the semiring is equipped with an order relation interpreted as usage subsumption or *compatibility*—a variable that is used p times can also be considered to be used q times if $q \leq p$. We proved key properties for usage such as a substitution lemma and subject reduction, and showed correctness of erasure, in the sense that parts marked as erasable can be safely removed during compilation.

Building on this work, we extend the formalization [2] with a correctness proof that goes beyond erasure: we prove that variables are used as many times as specified when a program is evaluated. Our approach is based on the work of Choudhury et al. [4] and involves a heap-based abstract machine with the capability to track how many times lookups may be performed. In contrast to their formulation, our machine uses an explicit stack of continuations. We also fix a problem with the usage relation in our previous work [1]: variable usage was sometimes handled incorrectly for `natrec`, the eliminator for natural numbers. As an example, our previous method allowed the program $\lambda^1 n.\texttt{plus}\ n\ n$ to be considered linear (here `plus` denotes addition for natural numbers, defined in a standard way).

In $\mathsf{natrec}^r_p\ (x_n.A)\ z\ (x_p.x_r.s)\ n$, the arguments z and s are the zero and suc branches, respectively, and n is the natural number argument. The s argument binds two variables, x_p, corresponding to the predecessor of the natural number, and x_r, corresponding to the recursive call. These are assigned the grades p and r, respectively ($p$redecessor and $r$ecursive call). There is also an explicit motive A dependent on the eliminated number (bound as x_n).

In order to find a correct usage count for `natrec` we make the following ansatz (which comes from following the structure of e.g. the eliminator for pairs):

$$\frac{\gamma_{\mathsf{z}} \blacktriangleright z \qquad \gamma_{\mathsf{s}}.\, x_p : p.\, x_r : r \blacktriangleright s \qquad \gamma_{\mathsf{n}} \blacktriangleright n}{q\gamma_{\mathsf{n}} + \delta \blacktriangleright \mathsf{natrec}^r_p\ (x_n.A)\ z\ (x_p.x_r.s)\ n}$$



This work is licensed under a Creative Commons "Attribution 4.0 International" license.

Here q is some grade indicating the number of copies of n that are used and δ is a context representing the usage contributions from z and s. The main difficulty in finding proper values for q and δ comes from the recursion: whatever usage counts we choose to assign, they should work for all values of n, i.e. regardless of how deep the recursion is.

Let δ_i be the usage contribution from z and s for the number i. First, δ_0 corresponds to the zero case where only z is used, so we get $\delta_0 = \gamma_\mathsf{z}$. Similarly, δ_{i+1} corresponds to a successor case which uses s once (with γ_s resources) and the recursive call r times (up to subsumption). A single recursive call uses δ_i resources, so we get $\delta_{i+1} = \gamma_\mathsf{s} + r\delta_i$. Now we let δ be the approximation that we get by taking the infimum of the δ_i (for $i \in \mathbb{N}$): if such an infimum does not exist then $\mathsf{natrec}^{r}_{p}\,(x_n.A)\,z\,(x_p.x_r.s)\,n$ is not considered to be well-resourced.

Analogously, we obtain q as the infimum of the sequence q_i, where $q_0 = 1$ since in the base case n is "consumed fully" by the match, and $q_{i+1} = p + rq_i$ since n is used p times by s and the recursive call is again used r times.

With this usage rule, we can show the same substitution, subject reduction and erasure correctness properties as before [1] (for semirings with well-behaved greatest lower bounds). However, it also allows us to show a more precise form of resource correctness, stated using an abstract machine with a heap and a stack.

A machine state $\langle H;\, t;\, \rho;\, S\rangle$ consists of a heap H, head t, environment ρ, and stack S. Environments map variable names to pointers (again, note that our formalization uses de Bruijn indices instead of variables, we have not proved that the rules involving variables below actually work). Heap entries $x \mapsto^p (t, \rho)$ are associated with a grade p, indicating how many times lookup may be performed. They also contain an environment ρ that maps the variable names of t to the pointers of the heap. The evaluation stack S is a list of *continuations* that represent the context "around" the term that is currently being evaluated (the head). Like entries, each continuation contains an environment mapping variables to pointers. Reduction is performed in a call-by-name style; these are the rules for the lambda calculus part:

$$\begin{aligned}
\langle H.\, \rho(x) \mapsto^p (t, \rho').\, H';\, x;\, \rho;\, S\rangle &\to \langle H.\, \rho(x) \mapsto^q (t, \rho').\, H';\, t;\, \rho';\, S\rangle && \text{if } p - |S| = q\\
\langle H;\, t\,^p u;\, \rho;\, S\rangle &\to \langle H;\, t;\, \rho;\, \bullet^p u[\rho].\, S\rangle\\
\langle H;\, \lambda^p x.t;\, \rho;\, \bullet^p u[\rho'].\, S\rangle &\to \langle H.\, y \mapsto^{|S|p} (u, \rho');\, t;\, \rho[x \mapsto y];\, S\rangle && y \notin H
\end{aligned}$$

For variables, a lookup is performed and the grade of the corresponding entry is updated, indicating that some allowed lookups have been "used". We do not require the semiring to be equipped with subtraction (i.e. to be a ring) and instead define a form of "partial" subtraction via the relation $p - q = r$, defined to hold if r is the least grade such that $p \leq r + q$ (we require that if there is such an r then there is a least one). Because an eliminator is not necessarily linear in its scrutinee, a given lookup might "consume" multiple copies. For instance, for $\mathsf{natrec}^{r}_{p}\,(x_n.A)\,z\,(x_p.x_r.s)\,n$ the grade q is assigned to the argument n (where q is the infimum of 1, $r+p$, r^2+rp+p, and so on), so the machine is set up so that when $\mathsf{natrec}^{r}_{p}\,(x_n.A)\,z\,(x_p.x_r.s)\,x$ is evaluated (in an empty stack) the lookup of x consumes q copies. The stack could contain several eliminators, so the variable rule uses $|S|$, a grade associated to all the eliminators in S (note that this grade might not be defined if some infimum does not exist).

The non-variable cases are mainly concerned with the stack. For eliminators like applications and natrec, a corresponding continuation is put on the stack and evaluation continues with the scrutinee. If the result is a compatible value, then the continuation is removed from the stack and zero or more new entries are added to the heap. The number of allowed lookups $|S|p$ for a new heap entry is given by a corresponding annotation p on the syntax, scaled by the stack multiplicity $|S|$.

We show termination: evaluation of a well-resourced term of type $\mathbb{N}$, starting with an empty heap and stack, will reach a final state with a numeral in head position and an empty stack. This implies that evaluation does not use variables more times than specified, since doing so is prevented by the reduction semantics. We also show that variables are not used fewer times than specified: the final heap does not contain any entries for which lookups are "necessary". For instance, if the linearity semiring is used, then there is no heap entry with the grade 1.

Acknowledgements. We would like to thank the reviewers. Andreas Abel and Oskar Eriksson acknowledge support by *Vetenskapsrådet* (the Swedish Research Council) via project 2019-04216 *Modal typteori med beroende typer* (Modal Dependent Type Theory). Oskar Eriksson additionally acknowledges support by *Knut and Alice Wallenberg Foundation* via project 2019.0116. Nils Anders Danielsson acknowledges support from Vetenskapsrådet (2023-04538).

References

[1] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. A graded modal dependent type theory with a universe and erasure, formalized. *Proc. ACM Program. Lang.*, 7(ICFP), August 2023. doi:10.1145/3607862.

[2] Andreas Abel, Nils Anders Danielsson, Oskar Eriksson, Gaëtan Gilbert, Ondřej Kubánek, Wojciech Nawrocki, Joakim Öhman, and Andrea Vezzosi. An Agda Formalization of a Graded Modal Type Theory with a Universe Hierarchy and Erasure, 2025. URL: https://github.com/graded-type-theory/graded-type-theory.

[3] Robert Atkey. Syntax and semantics of quantitative type theory. In *LICS '18: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 56–65, 2018. doi:10.1145/3209108.3209189.

[4] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434331.

[5] Conor McBride. I got plenty o' nuttin'. In *A List of Successes That Can Change the World*, volume 9600 of *LNCS*, pages 207–233, 2016. doi:10.1007/978-3-319-30936-1_12.

[6] Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded modal dependent type theory. In *Programming Languages and Systems, 30th European Symposium on Programming, ESOP 2021*, volume 12648 of *LNCS*, pages 462–490, 2021. doi:10.1007/978-3-030-72019-3_17.

Stellar - A Library for API Programming

André Videla

Glaive, Glasgow, UK
andre@glaive-research.org

Dependently typed programming language manipulate types, but types aren't enough to represent the general *interface* of programs. An Application Programming Interface (API) isn't defined using a type, but a *container* [2] instead. Defining APIs this way allows a declarative style of programming where manipulating APIs is the primary tool for building software. There already exist tools that represent APIs in software, for example the OpenAPI standard [1], or command-line interface DSLs. But we have no way of relating those two APIs. Containers and their morphisms give us the tools to talk about APIs in a principled ways and design complex software in a modular way.

APIs as Containers, What?

Containers defined as a pair of shape and positions can be reinterpreted as a pair of *query* and *response* ($query : Set \triangleright response : query \to Set$). With this interpretation, monoidal operators like coproducts ($+$), tensors ($\times$), composition ($\circ$), perfectly represent valid operations on APIs.

A coproduct of APIs $(query_1 \triangleright response_1) + (query_2 \triangleright response_2)$ constructs an API that gives the choice of what query to send $query_1 + query_2$ and return a response that matches the query that was send $[response_1, response_2]$.

A tensor of APIs $(query_1 \triangleright response_1) \times (query_2 \triangleright response_2)$ produces an API that expects both queries at once, and returns both response at the same time $(query_1 \times query_2 \triangleright response_1 \times response_2)$.

The composition of APIs $(query_1 \triangleright response_1) \circ (query_2 \triangleright response_2)$ defines a specific sequence of operations starting with $query_1$ and depending on its response, sending another query $query_2$, from which we derive the response for both $((q, k) : \Sigma(q : query_1).reponse_1(q) \to query_2 \triangleright \Sigma(r : response_1(q)).reponse_2(k(r))$

An example could help here

For sure! Let's say we are working with the movie database. It advertises a `GET` endpoint with path `/movie/{movie_id}` and this endpoint return responses `200 OK` with a JSON content body representing the movie's data. This API can be equally represented as the container $Info = (\texttt{movie\_id} \triangleright \texttt{Movie})$. Another endpoint `GET /movie/{movie_id}/alternative_titles` returns a list of titles given a movie id, so as a container it can be represented as $Titles = (\texttt{movie\_id} \triangleright \texttt{List Title})$. A server exposing this API gives any client the choice of what endpoint to call, and therefore, can be represented as the container $Info + Titles \equiv (\texttt{movie\_id} \triangleright \texttt{Movie}) + (\texttt{movie\_id} \triangleright \texttt{List Title})$.

What About Morphisms?

Morphisms of APIs tell us how to delegate from a higher-level API to a more specialised one. For example, a command line interface is an API represented with a container $CLI =$

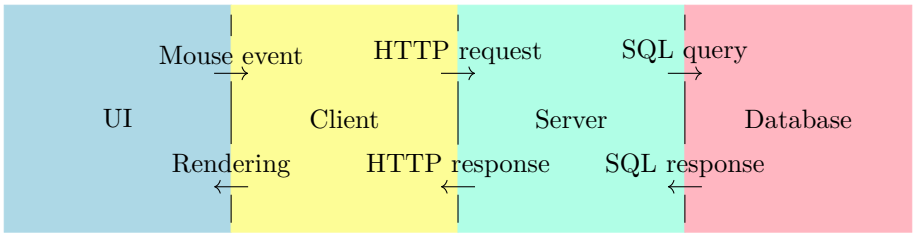


Figure 1: An illustration of a compositional pipeline of API morphisms. Containers are the vertical interstices between colors, the color blocks are container morphisms. The lifecycle of a single interaction can be read clockwise starting from the UI.

($List\ String \triangleright String$) indicating that it expects a list of string as input and returns a string to print as output. A morphism $CLI \twoheadrightarrow DB$ converts messages from $List\ String$ into database queries and, converts back responses from `Table` to `String` to print in the standard output.

Reusing the example of the movie database, if we want to create a program that exposes a command-line interface and forwards requests to the movie database, we could conceptualise this program as the morphism `CLI =>> Info + Titles`. Such morphisms in inhabited by two maps, one to convert command-line arguments to requests:

```
parseArgs : List String -> Movie_id + Movie_id
```

And one to convert http responses into strings we can display in the terminal:

```
convertResponses : Movie + List Title -> String
```

Those two functions can be combined in a single container morphism:

```
app : CLI =>> Info + Titles
app = !! \x => parseArgs x ## convertResponses
```

Surely This Can't Deal With Effects, Can It?

Yes it can! In fact there are many ways to express effectful computation with containers, the most common effects, non-determinism, and failure, can be represented as a List and Maybe monad [6] on containers.

The Maybe monad on container is defined by $Maybe(q \triangleright r) = (Maybe\ q \triangleright Any\ r)$ and the list monad on container is defined by $List(q \triangleright r) = (List\ q \triangleright All\ r)$.

Something like the previous mapping parsing command-line argument can now be written using the `Maybe` monad on containers to accurately represent the fact that parsing might fail: `app : CLI =>> Maybe (Info + Title)`.

Additionally, we can perform effect like IO by mapping the responses of a container with the $(\bullet)$ operation which applies a monad on types to turn it into a comonad on containers: $f \bullet (q \triangleright r) = (q \triangleright f \circ r)$. For example a program that prints its output can be seen as the morphism $IO \bullet (String \triangleright ()) \twoheadrightarrow (String \triangleright String)$ where the forward map is an identity and the backward map is given by `putStrLn : String -> IO ()`, this is an example of a monadic lens [3].

I Get the Idea, But How Do You Even Run Those Things?

A container morphisms which codomain is the monoidal unit can be converted into a function from its query to its response. The function $costate : (q \triangleright r) \twoheadrightarrow I \to (x : q) \to r(x)$ does exactly that. Those leaf morphisms represent computation that can be done immediately and are often found the end of a long sequence of API transformations converting APIs into increasingly simpler ones. Typically, sending an HTTP request amounts to implementing the morphism $HTTP \twoheadrightarrow I$, likewise for database queries and their responses $DB \twoheadrightarrow I$

Conclusion

Implementing containers in a dependently-typed programming language we can create a library which sole purpose is to describe API-level architecture. There are many use-cases for such a library and architecture, including server development, command-line tools, compilers, microservices, and more. We chose Idris [4, 5] for its package manager, ecosystem of low-level libraries, and flexible FFI.

References

[1] OpenAPI Specification v3.1.0 | Introduction, Definitions, & More.

[2] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of Containers. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Martin Hofmann, editors, *Typed Lambda Calculi and Applications*, volume 2701, pages 16–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[3] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on Monadic Lenses. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 1–31. Springer International Publishing, Cham, 2016.

[4] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, September 2013.

[5] Edwin Brady. Idris 2: Quantitative Type Theory in Practice. *arXiv:2104.00480 [cs]*, April 2021. arXiv: 2104.00480.

[6] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

An Inductive Universe for Setoids

Loïc Pujet

Stockholm University, Sweden
loic@pujet.fr

The setoid model of type theory was designed by Hofmann [5] in order to add function extensionality, proposition extensionality, and quotient types to intensional type theory. Unfortunately, Hofmann's construction does not support universes, which makes it impossible to even prove that $0 \neq 1$. Such advanced theorems had to wait for the work of Altenkirch [1], who improved the definition of setoids by putting the setoid equalities in a sort of *strict* propositions (`SProp`), and managed to equip the resulting model with a universe of propositions. A few years later down the line, Altenkirch, Boulier, Kaposi, Sattler and Sestini constructed a proper universe of setoids for Altenkirch's model [2]. They give three different definitions of their universe: one which needs *induction-recursion*, one which needs *induction-induction*, and finally one which only requires an identity type with a strengthened J rule.

In this abstract, we propose a new construction for the setoid universe which only needs indexed inductive families. Furthermore, we note that our construction still works (more or less) when the equality of the setoids is taken to be in `Prop` or `Type` instead of `SProp`, resulting in models that can be equipped with *choice principles*. Our construction is fully formalised in Rocq, and is available at https://github.com/loic-p/setoid-universe.

1 The Universe of Setoids

The most natural construction for the setoid universe is *via* induction-recursion: one defines a type of codes $\mathtt{U}_0$ along with three recursive functions $\mathtt{eqU}_0$, $\mathtt{El}_0$ and $\mathtt{eq}_0$ that respectively represent the setoid equality between the codes, the universal family of small setoids and its (heterogeneous) equality:

$$\mathtt{U}_0 : \mathtt{Type}_1 \qquad \mathtt{El}_0 : \mathtt{U}_0 \to \mathtt{Type}_0$$
$$\mathtt{eqU}_0 : \mathtt{U}_0 \to \mathtt{U}_0 \to \mathtt{Sort}_1 \qquad \mathtt{eq}_0 : \forall\ (\mathtt{A}\ \mathtt{B} : \mathtt{U}_0), \mathtt{El}_0\ \mathtt{A} \to \mathtt{El}_0\ \mathtt{B} \to \mathtt{Sort}_0$$

Note that we use an indeterminate sort $\mathtt{Sort}_i$ for the equality relations of our setoids, so that we may later instantiate it with either `SProp`, `Prop` or $\mathtt{Type}_i$. Anyway, we do not wish to use induction-recursion, so we must find a way to eliminate it from the construction. The canonical method is Hancock *et al.*'s small induction-recursion [4], but it does not apply in this case: not only do the recursive functions $\mathtt{eqU}_0$ and $\mathtt{eq}_0$ have two arguments of type $\mathtt{U}_0$ instead of one, but the return type of $\mathtt{eqU}_0$ is not even small (at least in the case where $\mathtt{Sort}_i = \mathtt{Type}_i$).

Instead, we start by defining an overapproximation $\mathtt{preU}_0$ for our setoid universe, with a constructor for dependent products that is parametrised by arbitrary equality relations on `A` and `P` and does not enforce `P` to be a setoid morphism (Fig. 1). This way, the definition of $\mathtt{preU}_0$ is not mutual with the definition of $\mathtt{eq}_0$ and $\mathtt{eqU}_0$ anymore, and it fits in the framework of small induction-recursion. Then, as a second step, we define the equality relations $\mathtt{eq}_0$ and $\mathtt{eqU}_0$ on the overapproximated universe, using the same definitions as in the usual inductive-recursive version. Next, we define an inductive predicate $\mathtt{extU}_0$ that carves out the codes from $\mathtt{preU}_0$ which have a counterpart in the inductive-recursive definition. More specifically, it ensures that that the dependent codomains that appear in $\mathtt{pre}_\Pi$ and $\mathtt{pre}_\Sigma$ are proper setoid morphisms from the domains into the universe, and that the codes for dependent products have

```
Inductive preU₀ : Type₁ :=
| preₙ : preU₀
| preΣ : ∀ (A : preU₀) (P : El₀ A → preU₀), preU₀
| preΠ : ∀ (A : preU₀) (Aeq : El₀ A → El₀ A → Sort₀)
          (P : El₀ A → preU₀) (Peq : ∀ a₀ a1, El₀ (P a₀) → El₀ (P a₁) → Sort₀), preU₀.

Fixpoint El₀ (A : preU₀) : Type₀ :=
 match A with
 | preₙ ⇒ ℕ
 | preΣ A P ⇒ Σ (a : El₀ A), El₀ (P a)
 | preΠ A Aeq P Peq ⇒ Σ (f : ∀ (a : El₀ A), El₀ (P a)), (∀ a a', Aeq a a' → Peq (f a) (f a'))
 end.
```

Figure 1: Small inductive-recursive definition of an overapproximated universe.

been parametrised with the equality relations defined by $\texttt{eq}_0$. Finally, we can put everything together: the carrier type of our universal setoid is defined as $\texttt{U}_0 := \Sigma\ (\texttt{A} : \texttt{preU}_0).\ \texttt{extU}_0\ \texttt{A}$, its setoid equality is given by $\texttt{eqU}_0$ on the first component, the universal dependent family on the universe is provided by $\texttt{El}_0$, and the heterogeneous equality on that family is given by $\texttt{eq}_0$. This roundabout encoding is actually faithful to the original inductive-recursive definition, as we can derive the same induction principle with its computation rules, and the three functions $\texttt{El}_0$, $\texttt{eqU}_0$ and $\texttt{eq}_0$ compute on type formers.

In order to complete the definition of our universal setoid, we can show by induction that the equality relation $\texttt{eqU}_0$ is an equivalence relation on $\texttt{U}_0$, and that the relation $\texttt{eq}_0$ is a heterogeneous equality equipped with a coercion operator:

```
cast   :  ∀ (A B : U₀) (e : eqU₀ A B) (a : El₀ A), El₀ B
casteq :  ∀ (A B : U₀) (e : eqU₀ A B) (a : El₀ A), eq₀ A B a (cast A B e a)
```

The accompanying Rocq development also includes W types, a subuniverse of propositions, an equality type, two universe levels, quotient types, and an accessibility predicate with its large elimination principle.

2 Choice principles

SProp setoids Instantiating the construction with `SProp` results in a universe that fits nicely in Altenkirch's setoid model, and does not need anything fancy from our metatheory. This provides a shallow embedding of MLTT + extensionality + quotients into MLTT + SProp, which preserves *all* the computation rules of MLTT. Note that even though the computation rule of the J eliminator for $\texttt{eq}_0$ is only propositional, one can nevertheless define an inductive equality in $\texttt{U}_0$ which is equivalent to $\texttt{eq}_0$ and for which J does compute on reflexivity [8].

Type setoids The situation is even more interesting if we try to instantiate the construction with `Type`. In that case, we can additionally interpret the following principle in our model, which says that any relation that is propositionally a functional relation determines a function (this principle is sometimes called *unique choice*):

```
∀ (a : A), ‖ Σ (b : B), (R a b) × (∀ c, R a c → c = b) ‖
  → Σ (f : A → B), ∀ (a : A), R a (f a).
```

And there's more: since the equality relation on the setoid of natural numbers coincides with the meta-theoretic equality on its underlying set, a function from $\mathbb{N}$ to any other setoid is automatically a setoid morphism. As a consequence, one can interpret countable choice and even dependent choice in this new model.

$$\begin{array}{lcl} \mathsf{AC}_{\mathbb{N}} & : & (\forall\ (\mathtt{n} : \mathbb{N}),\ \|\,\mathtt{P}\ \mathtt{n}\,\|) \ \to\ \|\,\forall\ (\mathtt{n} : \mathbb{N}),\ \mathtt{P}\ \mathtt{n}\,\| \\ \mathsf{DC} & : & (\forall\ (\mathtt{a} : \mathtt{X}),\ \|\,\Sigma\ (\mathtt{b} : \mathtt{X}),\ \mathtt{R}\ \mathtt{a}\ \mathtt{b}\,\|) \ \to\ \|\,\Sigma\ (\mathtt{s} : \mathbb{N} \to \mathtt{X}),\ \forall\ \mathtt{n},\ \mathtt{R}\ \mathtt{n}\ (\mathtt{n}{+}1)\,\| \end{array}$$

Unfortunately, there is a price to pay: if we try to do a setoid model without definitional proof irrelevance, the computation rules for substitution under binders do not hold definitionally. In particular, this means that our construction can no longer be viewed as a shallow embedding from an extension of MLTT to MLTT, since some definitional equalities are only interpreted as setoid equalities. This phenomenon has already been pointed out in a note of Coquand [3], in which he presents proof-relevant setoids as a model for an early version of MLTT which had weaker computation rules [6, 7].

Prop setoids If we instantiate our construction with `Prop` instead, we can have a sort of impredicative propositions without destroying the computational content of propositions. In particular, it allows us to have large elimination for the `Prop`-valued accessibility predicate. As a consequence, we can derive a version of unique choice which is restricted to decidable relations $\mathtt{R} : \mathtt{A} \to \mathbb{N} \to \mathtt{Prop}$. The resulting theory is extensional but keeps the proof-theoretic strength of CIC, which is much higher than the strength of MLTT+SProp. However, neither the full principle of unique choice nor countable choice are provable in this model, and the computation rules for binders are not definitional either.

References

[1] Thorsten Altenkirch. Extensional equality in intensional type theory. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 412–420, 1999.

[2] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, Christian Sattler, and Filippo Sestini. Constructing a universe for the setoid model. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures*, pages 1–21, Cham, 2021. Springer International Publishing.

[3] Thierry Coquand. About the setoid model. https://www.cse.chalmers.se/~coquand/setoid.pdf, 2013.

[4] Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small induction recursion. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications*, pages 156–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Martin Hofmann. A simple model for quotient types. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 216–234, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[6] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73 – 118. Elsevier, 1973.

[7] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In Stig Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundations of Mathematics*, pages 81–109. Elsevier, 1975.

[8] Loïc Pujet and Nicolas Tabareau. Observational Equality: Now For Good. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, January 2022.

Weihrauch problems as containers

Cécilia Pradic and Ian Price

Swansea University

Abstract

We note that Weihrauch problems can be regarded as containers over the category of projective represented spaces and that Weihrauch reductions correspond exactly to container morphisms. We also show that Bauer's extended Weihrauch degrees and the posetal reflection of containers over partition assemblies are equivalent. Using this characterization, we show how a number of operators over Weihrauch degrees, such as the composition product, also arise naturally from the abstract theory of polynomial functors.

The content discussed in this abstract is from [13].

Weihrauch Reducibility Weihrauch reducibility is a framework from computable analysis for comparing the computational strength of partial multi-valued functions over Baire space, i.e., relations on $\mathbb{N}^{\mathbb{N}}$, which are thus called *Weihrauch problems*. Intuitively, a problem P is reducible to a problem Q, written $P \leq_{\mathrm{W}} G$, if we can compute P given an oracle for Q that can be called once.

Definition 1 ([10])**.** *If P and G are two Weihrauch problems, P is said to be* Weihrauch reducible to *G if there exist partial type 2 computable*[1] *maps φ and ψ such that φ is a map $\mathrm{dom}(P) \to \mathrm{dom}(G)$ and for every $i \in \mathrm{dom}(P)$ and $j \in G(\varphi(i))$, $\psi(i,j)$ is defined and belongs to $P(i)$.*

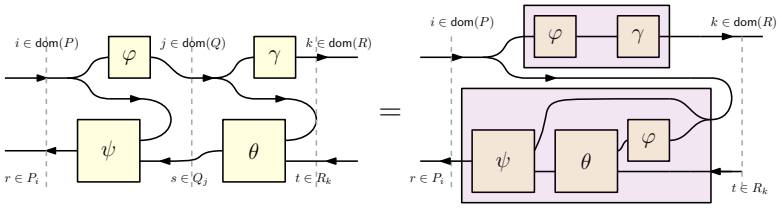
Example 1. *We can encode the following as Weihrauch problems:*

- LPO *("given a bit sequence, tell me if it has a 1"), defined by*

$$\mathsf{LPO}(p) = \{0^\omega \mid p = 0^\omega\} \cup \{1^\omega \mid p \in \{0,1\}^{\mathbb{N}}, p \neq 0^\omega\}$$

- KL*: "given an infinite finitely branching tree t, give me an infinite path through t" can also be encoded as a Weihrauch problem, modulo a standard embedding $2^{\mathbb{N}^{<\omega}} \hookrightarrow \mathbb{N}^{\mathbb{N}}$.*

LPO *is Weihrauch reducible to* KL*, but not* KL *is not reducible to* LPO*.*



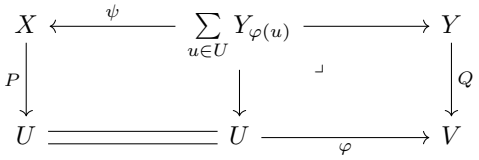
Since Weihrauch reductions compose as depicted above, Weihrauch problems and reductions form a preorder. The equivalence classes (called Weihrauch degrees) form a distributive lattice, where meets and joins can be regarded as natural operators on problems: the join $P \sqcup Q$ allows to ask a question either to P or Q and get the relevant answer, while $P \sqcap Q$ requires to ask one

[1] Type-2 computable maps are partial computable stream transformers $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$.

question to each and get only one answer. Many other natural operators have been introduced on the Weihrauch lattice, including a parallel product $P \times Q$ (ask questions to both P and Q, get both answers), a composition operator $P \star Q$ (ask a question to Q and then, depending on the answer you got, a question to P) and many others, including residuals and fixpoints of other operators [5, Definition 1.2].

Containers Intuitively, a container in a category $\mathcal{C}$ is a family of objects of $\mathcal{C}$ indexed by an object of $\mathcal{C}$. Since objects of $\mathcal{C}$ are not necessarily sets, to make this formal, we define *containers* as objects of $\mathcal{C}/_I$. Morphisms of containers are defined as follows.

Definition 2. *A morphism representative from a container $P : X \to U$ to $Q : Y \to V$ is a pair (φ, ψ) making the following diagram commute, the rightmost square being a pullback:*



A morphism of containers is an equivalence class of morphism representatives.

The category of containers crops up in several places in the literature on functional programming and mathematics and are sometimes referred to as polynomials or dependent lenses [1, 6, 17]. It is a "category of abundance", having four monoidal structures (products, coproducts, $\otimes$, and a composition $\circ$) and many other desirable properties [11, 14].

Contributions The representation of container morphisms into a "forward map" ϕ and a family of "backward maps" ψ, as well as similar types of algebraic structures in the poset of Weihrauch degrees and the category of containers suggests a strong relation between the two. Our main contribution is to make that connection formal: Weihrauch degrees are isomorphic to the posetal reflection of the full subcategory of *answerable* containers over the category $\mathsf{pMod}(\mathcal{K}_2^{\mathrm{rec}}, \mathcal{K}_2)$ of partitioned represented spaces and (type 2) computable functions. This is technically straightforward: one regards a container $P : X \to U$ as the problem of finding a section to P and morphisms as reductions between problems. P being answerable intuitively means that a (possibly non-computable!) section of P exists at all[2].

Thus a similar correspondence also applies to close variants such as continuous Weihrauch reducibility and extended Weihrauch degrees [3]. Our most recent ongoing work is characterising strong Weihrauch reducibility in a similar way via *dependent adaptors*[3].

Coproducts and products of containers correspond to the lattice operators in the Weihrauch degrees. The tensor product $\otimes$ of containers corresponds to the parallel product of Weihrauch degrees. The treatment of composition of containers and composition of Weihrauch degrees is trickier. The reason for this is that Weihrauch problem correspond to containers over a category which is only *weakly* (locally) cartesian closed. This means that we while we may define a composition operator, it will only be a quasi-bifunctor.

A summary of the current state of what we know and conjecture is given in table 1, for details see our preprint [13].

[2] In general, starting from any category $\mathcal{C}$ with pullbacks, we can say that P is answerable iff it is a pullback-stable epimorphism and derive the expected basic properties of $\times, +$ and $\otimes$ [13, §4.1].

[3] We adopt the definition and terminology from a seminar talk given by Hedges [4] – we are currently not aware of another citable source where the notion might have been spelled out.

Containers	Reducibility	Status
Answerable containers over $\mathsf{pMod}(\mathcal{K}_2^{\mathrm{rec}}, \mathcal{K}_2)$	Weihrauch problems	✓
Containers over $\mathsf{pAsm}(\mathcal{K}_2^{\mathrm{rec}}, \mathcal{K}_2)$	Extended Weihrauch problems	✓
Dependent adaptors over $\mathsf{pMod}(\mathcal{K}_2^{\mathrm{rec}}, \mathcal{K}_2)$	Strong Weihrauch problems	✓
Product $p \times q$	Meet $p \sqcap q$	✓
Coproduct $p + q$	Join $p \sqcup q$	✓
Tensor product $p \otimes q$	Parallel product $p \times q$	✓
Composition product $p \triangleleft q$	Composition product $p \star q$	✓*
Free monad on p	Iterated composition product $p^{\diamond}$	
Closed structures $\multimap, \Rightarrow$	?	
Derivative ∂p	?	
?	First-order part ${}^1p$	
?	Deterministic part $\mathrm{det}(p)$	
...	...	

Table 1: Relating container concepts to reducibility concepts.

Related work The idea of regarding a bundle as a problem to be solved by finding one of its section is an old one that predates the "container" terminology. For instance, Hirsch [7, Definition 3.4] defines an equivalent category to study the topological complexity of problems and reductions between those. This perspective also already appeared in the literature on Weihrauch problems (see for instance [9, Remark 2.8]), although most of the recent efforts we are aware of to "categorify" Weihrauch reducibility and operators on problems tend to use other tools instead.

One natural categorical construction that captured Weihrauch degrees that appeared in the literature is the restriction of Bauer's extended Weihrauch problems [3] to objects that are actually Weihrauch problems, which are characterized as the $\neg\neg$-dense predicates over modest sets. Interestingly, Ahman and Bauer also linked extended Weihrauch reducibility to containers [2], but by way of the more general notion of instance reducibility that works over families of truth values. while here we work directly with bundles of partitioned assemblies.

Trotta et. al. also formally linked (extended) Weihrauch reducibility with the Dialectica interpretation [15], which can be regarded bicompletions of fibrations by simple products and sums [8]. Aside from the fact that they work in a posetal setting throughout, it is interesting to note that the category of containers over $\mathcal{C}$ can be recovered by completing the terminal fibration over $\mathcal{C}$ by arbitrary products and then sums and taking the fiber over 1. The Dialectica interpretation was also used by Uftring [16] to capture Weihrauch reducibility in a syntactic way in a substructural arithmetic.

Pauly also studied a generic notion of reducibility that encompasses Weihrauch reducibility, starting from categories of multivalued functions [12], in which he derived the lattice operators as well as finite parallelizations in a generic way.

Acknowledgement We thank the anonymous reviewers, Arno Pauly and Takayuki Kihara for their comments on this work.

References

[1] Abbott, M.G., Altenkirch, T., Ghani, N.: Categories of containers. In: Gordon, A.D. (ed.) FOSSACS 2003 proceedings. Lecture Notes in Computer Science, vol. 2620, pp. 23–38. Springer (2003). https://doi.org/10.1007/3-540-36576-1_2

[2] Ahman, D., Bauer, A.: Comodule representations of second-order functionals (2024), https://arxiv.org/abs/2409.17664

[3] Bauer, A.: Instance reducibility and Weihrauch degrees. Log. Methods Comput. Sci. **18**(3) (2022). https://doi.org/10.46298/LMCS-18(3:20)2022, https://doi.org/10.46298/lmcs-18(3:20)2022

[4] Braithwaite, D., Capucci, M., Hedges, J., Gavranović, B., Rischel, E., Videla, A.: We solved dependent optics! (2025), https://www.youtube.com/watch?v=yhxwUnWKK2I, recording of a seminar talk given at the university of Strathclyde (MSP group) by Jules Hedges. Accessed on 06/05/2025.

[5] Brattka, V., Gherardi, G., Pauly, A.: Weihrauch complexity in computable analysis. CoRR **abs/1707.03202** (2017)

[6] Gambino, N., Kock, J.: Polynomial functors and polynomial monads. Mathematical Proceedings of the Cambridge Philosophical Society **154**(1), 153–192 (Sep 2012). https://doi.org/10.1017/s0305004112000394

[7] Hirsch, M.D.: Applications of topology to lower bound estimates in computer science. Ph.D. thesis, University of California, Berkeley (1990)

[8] Hofstra, P.J.W.: The Dialectica monad and its cousins. In: Makkai, M., Hart, B. (eds.) Models, Logics, and Higher-dimensional Categories: A Tribute to the Work of Mihály Makkai. CRM proceedings & lecture notes, American Mathematical Society (2011)

[9] Kihara, T.: Borel-piecewise continuous reducibility for uniformization problems. Logical Methods in Computer Science **12**(4) (Apr 2017). https://doi.org/10.2168/lmcs-12(4:4)2016

[10] Lempp, S., Miller, J., Pauly, A., Soskova, M., Valenti, M.: Minimal covers in the Weihrauch degrees. Proceedings of the American Mathematical Society **152**(11), 4893–4901 (2024)

[11] Niu, N., Spivak, D.I.: Polynomial functors: A mathematical theory of interaction (2024), https://arxiv.org/abs/2312.00990

[12] Pauly, A.: Many-one reductions between search problems. CoRR **abs/1102.3151** (2011), http://arxiv.org/abs/1102.3151

[13] Pradic, C., Price, I.: Weihrauch problems as containers (2025), https://arxiv.org/abs/2501.17250

[14] Spivak, D.I.: A reference for categorical structures on **Poly** (2024)

[15] Trotta, D., Valenti, M., de Paiva, V.: Categorifying computable reducibilities. Logical Methods in Computer Science **Volume 21, Issue 1**, 15 (Feb 2025). https://doi.org/10.46298/lmcs-21(1:15)2025, https://lmcs.episciences.org/11378

[16] Uftring, P.: The characterization of Weihrauch reducibility in systems containing $\mathrm{E-PA}^{\omega}$ + $\mathrm{QF-AC}^{0,0}$. The Journal of Symbolic Logic **86**(1), 224–261 (Oct 2020). https://doi.org/10.1017/jsl.2020.53, http://dx.doi.org/10.1017/jsl.2020.53

[17] Winskel, G.: Making concurrency functional (2023), https://arxiv.org/abs/2202.13910

Containers: Compositionality for Tensors

Neil Ghani[1], Pierre Hyvernat[2], and Artjoms Šinkarovs[3]

[1] Kodamai, Glasgow, Scotland. neil@kodamai.com
[2] Université Savoie Mont Blanc, Chambéry, France. pierre.hyvernat@univ-smb.fr
[3] University of Southampton, Southampton, UK. a.sinkarovs@soton.ac.uk

Abstract

While tensors are an important computational device, for example in machine learning [3] or physics [4], it seems type theory has had little to say about them. This work in progress is predicated on two principles widely accepted within the Types community: i) compositionality is a key approach enabling the construction of complex structures from simpler, and hence easier to define, structures; and ii) containers have an extremely rich compositional algebra [1]. Given these observations, one might wonder if containers can be used to develop a compositional approach to tensors and that is exactly what this abstract does.

1 Introduction

Tensors are actively used in many unrelated areas of computer science and mathematics. There are at least two ways to understand them. On the one hand, they are just multi-dimensional arrays, and this is the angle taken in array languages such as APL. On the other hand, tensors generalise matrices in the following way. If matrix is a normal form for linear functions, where matrix multiplication is function composition, tensors are normal forms for multi-linear maps from sets of vector and co-vector spaces into the underlying field. In both cases, tensors and their operations expose a lot of structure that is interesting to study in the context of type theory.

Despite their importance, tensor algebra tends to be seen as a tool with fairly concrete representation based on the manipulation of lists of natural number-valued indices — structure-preserving tensor operations are thus formalised via list-theoretic computation. While good for fast implementations, this approach suffers from a lack of high-level abstraction common to all modern development in type theory. As a result, proving properties of tensor inside systems such as Agda or Coq is rather cumbersome. We developed a container-based approach to tensors where properties of tensor operations are mapped onto well-known type-theoretic constructions. This is still a work in progress, but the clean and general treatment of, for example, reshaping via container morphisms, makes us confident this approach has something to bring.

Containers Containers [1] (also known as polynomial functors [2]) were introduced to study concrete data types. This is a powerful construction as it captures the notion of strictly positive data types, and it is closed under operations such as disjoint sum, product and many others. A container is a pair (S, P) where $S : \mathsf{Set}$ and $P : S \to \mathsf{Set}$. The set S is called the set of shapes and can be thought of as the constructors of a data type. Each constructor/shape $s \in S$ has an arity $P\ s$ which we call the positions of that shape. A container (S, P) is a presentation of the functor $[\![S, P]\!] : \mathsf{Set} \to \mathsf{Set}$ defined by

$$[\![S, P]\!]\ X = (\Sigma s : S) P\ s \to X$$

Elements in $[\![S, P]\!]\ X$ intuitively consist of the choice of a constructor and an assignment of a piece of data (here, X) to every position of that constructor. The fundamental theorem of

containers is a classification of the natural transformations between those: a natural transformation $f : [\![S, P]\!] \to [\![Q, R]\!]$ is uniquely given by: i) a function on shapes $u : S \to Q$; and ii) a contravariant re-indexing $t : (\Pi s : S)\ (R\ (u\ s) \to P\ s)$. Such pairs are called container morphisms and form a category Cont of containers.

Containers are known to support a large number of constructions making them good for compositional modelling. We use the coproduct which is defined as follows. Let (S, P) and (S', P') be containers. Their coproduct has shapes $S + S'$ with positions the cotuple $[P, P']$.

2 From Containers to Tensors

We change the usual perspective on containers by writing (A, I) to reflect i) $A : \mathsf{Set}$ will correspond to the set of axes (or dimensions) of a tensor; and ii) $I : A \to \mathsf{Set}$ assigns to a specific axis $a : A$, a set of indexes $I\ a$ on that axis. Consider a traditional two-dimensional $(n \times m)$-matrix which can be given as $A = 2$, $I\ (\mathsf{inl}\ *) = \mathsf{Fin}\ n$ and $I\ (\mathsf{inr}\ *) = \mathsf{Fin}\ m$. With $A : \mathsf{Set}$, there is no order on axes and hence we don't need the complication of re-ordering axes. Note that to get $(n \times m)$ matrices, the usual interpretation of containers needs to be changed:

$$[\![A, I]\!]_\Pi\ X = (\Pi\ A\ I) \to X$$

Note that $[\![-]\!]_\Pi : \mathsf{Cont} \to [\mathsf{Set}, \mathsf{Set}]$ is very different from the usual container interpretation. It is functorial and indeed $[\![-]\!]_\Pi = Y \circ \Pi$ where Y is the Yoneda embedding. While $[\![-]\!]_\Pi$ does not have the full range of compositional operators supported by $[\![-]\!]$, we do have: i) $[\![M]\!]_\Pi \circ [\![N]\!]_\Pi = [\![M + N]\!]_\Pi$; and ii) $[\![M]\!]_\Pi\ X \times [\![N]\!]_\Pi\ Y \to [\![M]\!]_\Pi\ (X \times Y)$. These properties give rise to the $\mathsf{pair}, \mathsf{nest}/\mathsf{unnest}, \mathsf{map}$ combinators:

$$\mathsf{pair} : [\![M]\!]_\Pi\ X \to [\![M]\!]_\Pi\ Y \to [\![M]\!]_\Pi\ (X \times Y)$$

$$\mathsf{nest} : [\![M + M']\!]_\Pi\ X \cong [\![M]\!]_\Pi\ ([\![M']\!]_\Pi\ X) : \mathsf{unnest} \qquad \mathsf{map} : (X \to Y) \to [\![M]\!]_\Pi\ X \to [\![M]\!]_\Pi\ Y$$

Reshaping is an important operation in array languages. Reshaping turns structural changes in shapes into actions on arrays, and it can be used to guide recursive traversals through array elements [5]. For tensors, reshaping can be given by container morphism and their action is defined as follows:

$$\mathsf{reshape} : \mathsf{Cont}(M, M') \to [\![M]\!]_\Pi\ X \to [\![M']\!]_\Pi\ X$$

Tensor contraction is one of the key operation in tensor calculus, and we can define this as follows:

$$\mathsf{matmul} : [\![M_1 + M_2]\!]_\Pi\ X \to [\![M_2 + M_3]\!]_\Pi\ X \to [\![M_1 + M_3]\!]_\Pi\ X$$

under the assumption that M_2 is finite and X is a ring. This operation also gives rise to the category of Tensors where objects are containers, morphisms from s to p are $[\![s + p]\!]_\Pi\ X$, and composition is given by matmul. Note that under this interpretation $[\![s + p]\!]_\Pi\ X$ is a tensor of s co-vectors and p (contravariant) vectors. Empty container $\mathbb{0}$ and the singleton container $\mathbb{1}$ give rise to singleton tensors which are usually referred as scalars. Any $[\![s]\!]_\Pi\ X$ can be turned into a "column vector" $[\![s + \mathbb{1}]\!]_\Pi\ X$ or a "row vector" $[\![\mathbb{1} + s]\!]_\Pi\ X$. Here distinction between vectors and co-vectors is not as strong as in the actual tensor calculus, but we have the structure to make it precise. Our work-in-progress formalisation can be found at https://github.com/ashinkarov/2025-types.

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. Applied Semantics: Selected Topics.

[2] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 210–225, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[3] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[4] P. Renteln. *Manifolds, Tensors and Forms*. Cambridge University Press, 2014.

[5] Artjoms Šinkarovs, Thomas Koopman, and Sven-Bodo Scholz. Rank-polymorphism for shape-guided blocking. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, FHPNC 2023, page 1–14, New York, NY, USA, 2023. Association for Computing Machinery.

Large Elimination and Indexed Types in Refinement Types

Alessio Ferrarini[1] and Niki Vazou[2]

[1] IMDEA Software Institute, Madrid, Spain alessio.ferrarini@imdea.org
[2] IMDEA Software Institute, Madrid, Spain niki.vazou@imdea.org

Abstract

We explore how to approximate large elimination and indexed datatypes in refinement type systems with a non-dependent base type system.

Introduction. At the core of modern dependent type theories [2], indexed inductive types and large elimination are the main means of how new datatypes are introduced. They also play a vital role in the kind of theorems that are provable in our theory. As an example, removing large elimination from MLTT [4] causes the loss of the ability to prove the disjointness of constructors [5]. This makes the theory unable to prove trivial theorems; for example, we cannot show that $\Pi(Eq(Bool, true, false), Void)$ is inhabited.

Refinement types as implemented in Liquid Haskell [8, 9] extend the type system of Haskell, SystemFC, with logical predicates. These predicates are Haskell (terminating) boolean expressions and operators that belong to the decidable fragment of SMT-Lib.

As refinement type systems lack direct support for large elimination and indexed datatypes, their absence makes it challenging to encode certain proofs and datatype definitions. In this work, we explore how to approximate these features within the refinement type system. As a running example, we translate the correct compiler of STLC to the SKI calculus presented in [6] from Agda [7] to Liquid Haskell.[1]

```
data Term : Ctx → Ty → Set where
  app : ∀ {Γ σ τ} → Term Γ (σ ⇒ τ)
      → Term Γ σ → Term Γ τ
  lam : ∀ {Γ σ τ} → Term (σ :: Γ) τ
      → Term Γ (σ ⇒ τ)
  var : ∀ {Γ σ} → Ref σ Γ → Term Γ σ
```

(a) Agda definition

```
data Term where
{-@ App :: γ:Ctx → σ:Ty → τ:Ty
        → Prop (Term γ (Arrow σ τ))
        → Prop (Term γ σ)
        → Prop (Term γ τ) @-}
  App :: Ctx → Ty → Ty → Term → Term
      → Term
{-@ Lam :: γ:Ctx → σ:Ty → τ:Ty
        → Prop (Term (Cons σ γ) τ)
        → Prop (Term γ (Arrow σ τ)) @-}
  Lam :: Ctx → Ty → Ty → Term
      → Term
{-@ Var :: γ:Ctx → σ:Ty
        → Prop (Ref σ γ)
        → Prop (Term γ σ) @-}
  Var :: Ctx → Ty → Ref → Term
data TERM = Term Ctx Ty
```

(b) Liquid Haskell definition

Figure 1: Well-typed STLC terms definition

[1] The full translation can be found at https://github.com/ucsd-progsys/liquidhaskell/blob/develop/tests/ple/pos/SKILam.hs.

```
data Ty : Set where
    ι   : Ty
    _⇒_ : Ty → Ty → Ty

Value : Ty → Set
Value ι       = ℤ
Value (σ ⇒ τ) = Value σ → Value τ
```

(a) Agda definition

```
data Value where
{-@ VIota :: Int → Prop (Value Iota) @-}
  VIota :: Int → Value
{-@ VFun :: σ:Ty → τ:Ty
         → (Prop (Value σ) → Prop (Value τ))
         → Prop (Value (Arrow σ τ)) @-}
  VFun :: Ty → Ty → (Value → Value)
       → Value
data VALUE = Value Ty
```

(b) Liquid Haskell definition

Figure 2: HOAS representation of values

Indexed Inductive Datatypes. A first solution for the absence of indexed inductive datatypes was proposed in [1] under the name of *data proposition*. The idea is to encode the information and constraints that in the dependently typed world is carried through indexes by refining the constructors of the datatype. In Liquid Haskell, this role is taken by the Prop type, which is a type alias for the refined type: `type Prop e = {v:a | e== prop v. Here, prop is treated as an uninterpreted function in the`logic and `e` is substituted by the index information. Since the logic has no knowledge about `prop` other than its type, two types can match only if the arguments passed to `Prop` are the same. In Figure 1, we see side by side the datatypes that encode well-typed terms of the simply typed lambda calculus.[2]

Large elimination. Large elimination instead is trickier as the natural solution would be to encode it through functions, but unfortunately this is not feasible as functions need to remain valid Haskell code and types are not part of valid Haskell expressions. A solution could be to use codes to encode types through codes mimicking the universe level encoding in dependent types but at the Haskell level there is no way to have a Haskell type depend on a value. The only possible solution then is to use a datatype declaration indexed by the argument of the dependent elimination where we have a constructor for each case and in each constructor we wrap a value of the type returned by the elimination. However this leads to problem when we construct an arrow type, as the same type will appear in a negative position, and it will fail to type check. But are these type declaration actually problematic? The type definition is actually well-founded as the index is structurally decreasing, otherwise the same declaration would be rejected by Agda. Thus, at least, the usual example showing that negative types lead to inconsistency does not apply. Another feature obtained via large elimination is *non-uniform dependencies*, i.e., functions are able to inspect their argument and change their type accordingly. The usual example is the type safe `sprintf`, but this kind of pattern can still be emulated through indexed families even if in a less elegant manner through sized lists.

In Figure 2 we compare the HOAS representation of values. In the Liquid Haskell translation, the `VFun` constructor has the negative occurrences issue.

Completing the Connection. Naturally, we can ask if this transformation always works and what is the relationship between the original type and its refined encoding? If these questions have satisfactory answers, we would then ask if any statement encodable with dependent types

[2] In the (Liquid) Haskell definition we use γ instead of Γ to denote contexts as as variable names cannot begin with capital letters.

can be translated into refinement types. The work in [3] shows that $ATTT$, a variant of System F with refinements, from an expressiveness perspective corresponds to second-order logic. However, it is unclear whether the notion of refinement in $ATTT$ aligns with that of Liquid Haskell.

Acknowledgments. Partially funded by the European Union (GA 101039196). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the European Research Council can be held responsible for them.

References

[1] M. H. Borkowski, N. Vazou, and R. Jhala. Mechanizing Refinement Types. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 2024.

[2] T. Coquand and C. Paulin. Inductively defined types. *Lecture Notes in Computer Science*, 1990.

[3] S. Hayashi. Logic of refinement types. In *Types for Proofs and Programs*, 1994.

[4] P. Martin-Löf. An intuitionistic theory of types: Predicative part. *Logic Colloquium*, 1975.

[5] J. M. Smith. The independence of peano's fourth axiom from martin-löf's type theory without universes. *The Journal of Symbolic Logic*, 1988.

[6] W. Swierstra. A correct-by-construction conversion from lambda calculus to combinatory logic. *Journal of Functional Programming*, 2023.

[7] The Agda Development Team. *Agda wiki*, 2024.

[8] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2014.

[9] N. Vazou, A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala. Refinement Reflection: Complete Verification with SMT. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.

HoTTLean: Formalizing the Meta-Theory of HoTT in Lean

Joseph Hua[1], Steve Awodey[1], Mario Carneiro[2],
Sina Hazratpour[3], Wojciech Nawrocki[1], Spencer Woolfson[1], and Yiming Xu[4]

[1] Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
{jlhua,awodey,wnawrock,swoolfso}@andrew.cmu.edu
[2] Chalmers University of Technology, Gothenburg, Sweden
marioc@chalmers.se
[3] Stockholm University, Stockholm, Sweden
sinahazratpour@gmail.com
[4] Ludwig Maximilian University of Munich, Munich, Germany
yiming.xu@lmu.de

Introduction. While elegant synthetic proofs such as those in *Homotopy Type Theory* (HoTT) are expected to "compile" to proofs of classical theorems when interpreted in suitable models, this idea has not yet been exploited in proof assistants. For example, Cubical Agda supports synthetic reasoning about cubical types, but its proofs have not been translated formally to facts about cubical sets, let alone their topological realizations. The HoTTLean project aims to bridge this gap by formalizing in Lean the semantics of a type theory we call HoTT0, a fragment of HoTT where univalence holds only on set-truncated types. We define the class of natural models [Awo18] of HoTT0, and prove that its syntax has a sound interpretation in any such model. As one concrete instance, we formalize the groupoid model of HoTT0 [HS98], providing a specific "compilation target" for synthetic proofs. Finally, we work toward embedding HoTT0 as a domain-specific language using Lean's extensible syntax and meta-programming facilities [UdM20]. Overall, this allows users to write synthetic proofs in HoTT0 and use the interpretation to produce constructions pertaining to groupoids as defined in Mathlib, the standard library for mathematics in Lean [Com20]. Through its compositional and modular approach, our project not only provides a bridge between synthetic and classical mathematics, but also lays the foundations for formalized semantics of other internal languages such as (complete) HoTT [Uni13] or directed type theory [Nor19].

Project structure. This project consists of the following components:

- **Type theory.** The type theory HoTT0 is a variant of that described in the HoTT book [Uni13]. HoTT0 has N Russell-style universes $U_1 : \cdots : U_N$. There can only be finitely many universes due to Gödelian constraints. HoTT0 has Σ-types, Π-types, and intensional identity types. We define subuniverses of set-truncated types internally in HoTT0, and require univalence axioms for each of these subuniverses. Though univalence implies functional extensionality in the subuniverses, to get functional extensionality for all types we postulate it as another axiom.

  In our Lean formalization, we present HoTT0 by an inductive type `Expr` of raw terms (quotient-inductive-inductive types [ACD+18] are not available to us) together with proof-irrelevant typing judgments `EqTp` and `EqTm` specifying type and term equality, respectively. We view these as partial equivalence relations, defining $\Gamma \vdash t : A \triangleq \Gamma \vdash t \equiv t : A$ and correspondingly for typehood. To reduce proof burden, we build presuppositions into some of the inference rules, as well as postulate rather than prove symmetry, transitivity, and closure under substitution. We establish only very basic syntactic metatheorems, preferring to work with the semantics to the maximal extent possible.

- **Interpretation.** Natural model semantics were developed by Awodey [Awo18], with additional simplifications in the definition of identity types suggested by Richard Garner. The construction makes use of polynomial functor machinery, which is currently being formalized in the parent project Poly, available at github.com/sinhp/Poly.

  We define the class of HoTT0 natural models, and show that HoTT0 syntax has a sound interpretation into any such model. Thanks to this compartmentalization, our project could be extended with other model constructions (such as simplicial sets) without altering the syntax or having to reprove soundness.

  In our Lean formalization, a model of HoTT0 is a sequence of `NaturalModelBase` structures connected by *universe morphisms* `UHom`, together with additional data to support type constructors. Each morphism provides the semantics for one universe. We construct an interpretation of HoTT0 into any `UHomSeq` as a partial function defined by recursion on raw terms. We then show that this function is total on well-formed types and terms, and respects judgmental equality.

- **Groupoid model.** In our Lean formalization, the category `Ctx` of contexts is the category of 1-groupoids `Grpd.{N+1,N+1}` with `Type (N+1)`-sized objects and morphisms (as defined in Mathlib). This supports a natural model structure with N universes, Σ, Π, and identity types. Semantically, univalent subuniverses of set-truncated types correspond to subcategories of discrete groupoids.

  The type classifier `Ty` is defined to be the presheaf on `Ctx` that takes a context Γ to the set of functors $\Gamma \to$ `Grpd.{N+1,N+1}`. Up to isomorphism, a type $\Gamma \to$ `Ctx` corresponds to an isofibration of groupoids, but this view does not provide a strict interpretation of the syntax (substitution equations only hold up to isomorphism). The interpretation of Σ-types is thus not simply defined using the composition of isofibrations. For each piece of syntax, we explicitly describe the action on the classifiers as natural transformations between presheaves.

  Terms are classified by the presheaf that takes a context Γ to the set of functors $\Gamma \to$ `PGrpd.{N+1,N+1}` where `PGrpd.{N+1,N+1}` is the category of *pointed* groupoids and functors preserving the point up to isomorphism. Context extension is defined using the Grothendieck construction `Grothendieck` from Mathlib.

- **Embedded proof assistant.** An aim of this project is to demonstrate that internal proofs in HoTT0 can be translated into proofs of theorems about Mathlib's groupoids, with the proof assistant offering automated translation. We are working on an embedded HoTT0 proof mode with support for seamless transitions between reasoning internally and reasoning about the groupoid model. Thanks to Lean's extensible syntax, macro system, and elaboration facilities [UdM20], we may reuse existing tactics and user interface elements in the HoTT0 proof mode, providing Lean users with a familiar interactive environment.

Formalization progress. HoTTLean is work in progress. The repository is accessible at sinhp.github.io/groupoid_model_in_lean4. So far we have constructed a fragment of the syntax with universes, Σ and Π-types and an interpretation of this syntax into its corresponding natural model semantics. For the groupoid model, we have constructed semantics for universes and Σ-types, with Π-types in the making.

Related work. We know of one other project that worked towards a formalization of the groupoid model of HoTT, by Sozeau and Tabareau [ST14]. Although their groupoid semantics follow the style of categories with families, they did not formalize categories with families as a class of models. Meanwhile, our project develops abstract natural model semantics, and then constructs the groupoid model as a particular instance. Another key difference is that their groupoids are enriched in setoids, while ours are not. Finally, a novelty of HoTTLean is in building a domain-specific proof mode that allows users to develop formal internal language arguments.

Ahrens, North, and van der Weide formalized the semantics of bicategorical type theory in Coq [ANvW23]. Unlike HoTTLean, they did not formalize the syntax or its interpretation, nor did they consider Σ and Π-types.

Maillard and Xu are constructing a deep embedding of geometric logic in Lean that they unfold into presheaf semantics in order to obtain theorems about Mathlib's algebraic structures [XM25].

The Flypitch project of Han and van Doorn established the independence of the continuum hypothesis in Lean [HvD20]. Mathlib now contains model and set theory libraries, and methods similar to ours could be employed to facilitate (often syntactically complex) internal arguments there.

Future work.

- HoTT0 can and should be extended to offer a class of higher inductive types. Possible candidates that are consistent with groupoid semantics include higher W-types [Vid18] and groupoid quotients [VvdW21]. From groupoid quotients one could construct the classifying space BG of a group G, as well as 0-truncations (used to define homotopy groups).

- Here we focus on semantic foundations, but one could use HoTT0 to develop univalent set-level mathematics. Univalence for sets can be used to implement the Structure Identity Principle, allowing for rigorous identification of isomorphic structures [Acz11].

- It may be possible to integrate our work with `Ground Zero` (github.com/forked-from-1kasper/ground_zero), a recent HoTT library using Lean 4.

This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-21-1-0009.

References

[ACD+18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. *Quotient Inductive-Inductive Types*, page 293–310. Springer International Publishing, 2018.

[Acz11] Peter Aczel. On Voevodsky's univalence axiom. *Mathematical Logic: Proof Theory, Constructive Mathematics, Samuel R. Buss, Ulrich Kohlenbach, and Michael Rathjen (Eds.). Mathematisches Forschungsinstitut Oberwolfach, Oberwolfach*, page 2967, 2011.

[ANvW23] Benedikt Ahrens, Paige Randall North, and Niels van der Weide. Bicategorical type theory: semantics and syntax. *Mathematical Structures in Computer Science*, 33(10):868–912, 2023.

[Awo18] Steve Awodey. Natural models of homotopy type theory. *Mathematical Structures in Computer Science*, 28(2):241–286, 2018.

[Com20] The Mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[HS98] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

[HvD20] Jesse Michael Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, 2020.

[Nor19] Paige Randall North. Towards a directed homotopy type theory. *Electronic Notes in Theoretical Computer Science*, 347:223–239, 2019.

[ST14] Matthieu Sozeau and Nicolas Tabareau. Towards an internalization of the groupoid model of type theory. *Types for Proofs and Programs 20th Meething (TYPES 2014), Book of Abstracts*, 2014.

[UdM20] Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 167–182, Cham, 2020. Springer International Publishing.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[Vid18] Jakob Vidmar. *Polynomial functors and W-types for groupoids*. PhD thesis, University of Leeds, 2018.

[VvdW21] Niccolò Veltri and Niels van der Weide. Constructing higher inductive types as groupoid quotients. *Logical Methods in Computer Science*, Volume 17, Issue 2, April 2021.

[XM25] Yiming Xu and Kenji Maillard. Geometric reasoning in Lean: from algebraic structures to presheaves. In *31st International Conference on Types for Proofs and Programs*, 2025.

Towards Formalising the Guard Checker of Rocq

Yee-Jian Tan[1,2] and Yannick Forster[1]

[1] Inria Paris, France
[2] Institut Polytechnique de Paris, France

Abstract

Rocq's consistency — and the consistency of constructive type theories in general — crucially depends on all recursive functions being terminating. Technically, in Rocq, this is ensured by a so-called guard checker, which is part of Rocq's kernel and ensures that fixpoints defined on inductive types are structurally recursive. Rocq's guard checker was first implemented in the 1990s by Christine Paulin-Mohring, but was subsequently extended, adapted, and fixed by several contributors. As a result, there is no exact, abstract specification of it, meaning for instance that formal proofs about it are out of reach. We propose a talk on a first step preceeding the formalisation of the guard checker, namely, a faithful implementation of the guard checker of Rocq in Rocq, using the MetaRocq framework. We hope that our project will benefit the users of Rocq by providing an accurate and transparent implementation, as well as lay the foundations for future formalisation efforts or even mechanised consistency proofs of Rocq.

In Rocq, one can define inductive types which have parameters, are indexed, mutual, or nested. Common examples are natural numbers, lists (which have a parameter), vectors (which have indices), even/odd predicates (which are mutual), the accessibility predicate (which has nesting via a function type), and rose trees (which have nesting via inductive types). One can then define so-called fixpoints by structural recursion on elements of these inductive types. As an example here is a definition of the recursor on natural numbers as a fixpoint:

```
Fixpoint nat_rec (P : nat → Set) (p0 : P 0) (ps : ∀ (m: nat), P m → P (S m))
  (n : nat) {struct n}: P n :=
   match n with
    | 0 ⇒ p0
    | S m ⇒ ps m (nat_rec P p0 ps m)
   end.
```

All the mentioned features like mutuality and nesting introduce mostly orthogonal complexity in the definition of Rocq's guard checker, which is defined in the `kernel/inductive.ml` file of Rocq's code in about 1000 lines of OCaml code. To illustrate how a guard checker is crucial for consistency, one can look at the following two non-terminating fixpoints

```
Unset Guard Checking.
Fixpoint boom (x : nat) {struct x} : False := boom x.

Fixpoint inf (n : nat) {struct n} :=
  match n with
   | 0 ⇒ 0
   | S _ ⇒ S (inf n)
  end.
```

They allow proofs of `False` as `boom 0` and exploiting the contradictory property of `inf` $1 =$ `S` (`inf` 1).

History of the guard condition In the 1990s, Frank Pfenning and Christine Paulin-Mohring introduced inductive types in the calculus of constructions [15] along with a guard condition for fixpoints [14]. The first version of Rocq's Guard Checker was implemented in Rocq v5.10.2 by Paulin-Mohring in 1994 [5]. In 2012, Pierre Boutillier relaxed the guard checker via β-ι commutative cuts [3]. In 2014, Maxime Dénès restricted the guard checker to forbid an unwanted proof that propositional extensionality does not hold in Rocq [7]. In 2022, Hugo Herbelin restricted the guard checker to ensure strong normalisation rather than

just weak normalisation, which is a behaviour that seems to be more in line with the intuition of users [9]. In 2024, Herbelin introduced a relaxation, allowing simpler implementation of nested recursive functions [10]. Furthermore, changes to the guard condition keep being proposed, see e.g. [11] and [12].

MetaRocq: a formalisation of Rocq in Rocq Two central parts of the MetaRocq project [16] are a formalisation of Rocq's type theory in Rocq [17] and, on top of that, verified implementations of a type checker [18] and a verified extraction function to OCaml [8]. The formalisation of type-theory faithfully captures typing rules of Rocq in an inductive predicate, and is parameterised in a guard checker function which is required to fulfil some basic properties such as being stable under reduction and substitution. Based on this formalisation, crucial properties such as subject reduction (types are preserved by reduction) and canonicity (normal forms of inductive types start with a constructor) are proved [18].

The verified type checker axiomatically assumes that reduction in the system is strongly normalising. The strong normalisation assumption can also be used to prove consistency, because any proof of `False` would have a normal form using strong normalisation, which would have the same type using subject reduction, and would start with a constructor of the inductive type `False` using canonicity – which is a contradiction, because `False` has no constructors.

An implementation of Rocq's Guard Checker in Rocq In the talk, we will report on the current state of the project: a full and faithful implementation of Rocq's guard checker in Rocq using the MetaRocq project [19], whose code can also be found online. We will also explain the workings of the guard checker via the data structures it uses and present its different dimensions of complexity, due to its many contributions by different authors.

Towards a verified guard checker As future work, the first step to verifying the guard checker would be synthesising a guard condition predicate from the current OCaml implementation of the guard checker. This predicate will work similarly to the typing predicate in MetaRocq, i.e. talk about the syntax of Rocq as specified in MetaRocq. This specification can be an inductive predicate, meaning it does not have to be obviously decidable.

As a second step, a guard checker function deciding the guard condition predicate should be defined. This function will have to use MetaRocq's verified implementation of reduction, meaning it will have to rely on the strong normalisation axiom. Technically, a substantial challenge will lie in proving that the definition of this guard checker function, defined in Rocq and working on syntax as specified by MetaRocq, is terminating. That this function indeed correctly computes the guard condition predicate can either be immediately proved by using dependent types and a correct-by-construction guard checker function, or in a separate additional step. Note that technically, proving the soundness of the function suffices, i.e. that whenever the function accepts a term, the guard condition predicate holds.

Towards normalisation proofs Having a formal description of the guard condition now allows relative normalisation proofs of Rocq's type theory in Rocq. Namely, it then is feasible to define a simpler, more natural, and more modular variant of the guard condition and show that Rocq's type theory with the current implementation of the guard condition can be interpreted in this simpler theory. This would result in a relative normalisation proof, meaning that Rocq's type theory is normalising (and thus consistent) if the simpler system is normalising. In particular, this means that the trust in Rocq's consistency could be moved to trusting that such a simpler theory is terminating. In the long term future, it then even could be possible to *prove* termination of Rocq's type theory (for a restricted number of universes to get around Gödel incompleteness issues) by reducing it in many steps to an extension of the currently also ongoing formalisation of MLTT in Rocq [2].

Related Work Termination checking in Agda is done semantically via sized types: a special, implicit type used by the type checker to determine if the recursion done on a strictly smaller argument [1, 13]. Lean, on the other hand, only has recursors (or eliminators) in its type theory, thus user-written recursive functions are represented in the kernel by recursors [4, 6].

References

[1] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *J. Funct. Program.*, 12(1):1–41, 2002. doi:10.1017/S0956796801004191.

[2] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. Martin-löf à la coq. In Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, pages 230–245. ACM, 2024. doi:10.1145/3636501.3636951.

[3] Pierre Boutillier. A relaxation of Coq's guard condition. In *JFLA - Journées Francophones des langages applicatifs - 2012*, pages 1 – 14, Carnac, France, February 2012. URL: https://hal.science/hal-00651780.

[4] Joachim Breitner. Reference for equations compiler using brecon. https://leanprover.zulipchat.com/#narrow/stream/270676-lean4/topic/Reference.20for.20equations.20compiler.20using.20brecOn/near/465564128, 2024. URL: https://leanprover.zulipchat.com/#narrow/stream/270676-lean4/topic/Reference.20for.20equations.20compiler.20using.20brecOn/near/465564128.

[5] Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, César Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The coq proof assistant-reference manual. *INRIA Rocquencourt and ENS Lyon, version*, 5, 1996.

[6] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

[7] Maxime Dénès. Tentative fix for the commutative cut subterm rule. https://github.com/coq/coq/commit/9b272a861bc3263c69b699cd2ac40ab2606543fa, 2014. URL: https://github.com/coq/coq/commit/9b272a861bc3263c69b699cd2ac40ab2606543fa.

[8] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. Verified extraction from coq to ocaml. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024. doi:10.1145/3656379.

[9] Hugo Herbelin. Check guardedness of fixpoints also in erasable subterms. https://github.com/coq/coq/pull/15434, 2022. URL: https://github.com/coq/coq/pull/15434.

[10] Hugo Herbelin. Extrude uniform parameters of inner fixpoints in guard condition check. https://github.com/coq/coq/pull/17986, 2024. URL: https://github.com/coq/coq/pull/17986.

[11] Hugo Herbelin. How much do system t recursors lift to dependent types? 2024. URL: https://types2024.itu.dk/abstracts.pdf.

[12] Hugo Herbelin. Size-preserving dependent elimination. 2024. URL: https://types2024.itu.dk/abstracts.pdf.

[13] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92. ACM, 2001. doi:10.1145/360204.360210.

[14] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types. (Inductive Definitions in Type Theory)*. 1996. URL: https://tel.archives-ouvertes.fr/tel-00431817.

[15] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989. URL: https://doi.org/10.1007/BFb0040259, doi:10.1007/BFB0040259.

[16] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The metacoq project. *J. Autom. Reason.*, 64(5):947–999, 2020. URL: https://doi.org/10.1007/s10817-019-09540-0, doi:10.1007/S10817-019-09540-0.

[17] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020. doi:10.1145/3371076.

[18] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo

Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. working paper or preprint, April 2023. URL: https://inria.hal.science/hal-04077552.

[19] Yee-Jian Tan and Yannick Forster. Towards Formalising the Guard Checker of Coq. Technical report, Ecole polytechnique ; Inria - Paris, March 2025. URL: https://inria.hal.science/hal-04983786.

Lean4Lean: Mechanizing the Metatheory of Lean

Mario Carneiro

Chalmers University of Technology
University of Gothenburg
Gothenberg, Sweden
marioc@chalmers.se

Introduction. The Lean proof assistant [2] is seeing increasing use in mathematics formalization and software verification, but the metatheory of Lean is not yet completely understood. In particular, while there is an intended interpretation within a classical set-theoretical model ([1], based on [5]), the proof of soundness has not been formalized. The `Lean4Lean` project endeavors to verify the key theorems (and non-theorems!) of the LeanTT formal system. Additionally, `Lean4Lean` provides an executable typechecker, written in Lean itself. This is the first complete typechecker for Lean 4 other than the reference implementation in C++ used by Lean itself, and our new checker is competitive with the original, running between 20% and 50% slower and usable to verify all of Lean's Mathlib library, forming an additional step in Lean's aim to self-host the full elaborator and compiler. Ultimately, we plan to use this project to help justify any future changes to the kernel and type theory and ensure unsoundness does not sneak in through either the abstract theory or implementation bugs.

The project is most directly comparable to the MetaRocq project [3, 4], and aims to do the same things that it accomplishes for the Rocq theorem prover, but for Lean's type theory. This is complicated by the fact that LeanTT is known to not be theoretically optimal: in particular, the typing relation is undecidable and Lean underapproximates it, and reduction is not strongly normalizing. (Canonicity also fails for the basic reason that LeanTT includes 3 axioms,[1] one of which is the axiom of choice.)

Project structure. The `Lean4Lean` project consists of three major components:

- **Executable typechecker.** This is a "carbon copy" reimplementation of the C++ Lean kernel written in Lean. It is written in monadic style, using only pure functions (i.e. using some combination of state, reader and exception monads rather than `IO`), and is designed to be efficiently compiled by the Lean compiler so that the resulting binary is competitive with the original. Because Lean is a functional programming language with limited optimizations, there is still a performance gap of about 20–50% compared to the C++ version, but this may improve along with the Lean compiler. This is already good enough to be usable as an additional or alternative checking step for Lean projects, and we would like to make a case for it to *replace* the C++ kernel.[2]

- **Formalized metatheory.** This is the most interesting part of the project for type theorists, and amounts to a definition of a type `VExpr` representing expressions (extrinsically typed, using pure de Bruijn variable convention), a judgment $\Gamma \vdash e \equiv e' : \tau$ which deals

[1] The kernel actually implements a theory LeanTT_0 which lacks these axioms but this system is not as well studied. LeanTT adds to LeanTT_0 the axioms `propext`, `Quot.sound`, and `choice`, and the main metatheory, including the soundness proof, targets this extended system.

[2] The actual decision for if and when this happens is under the control of the Lean FRO, so I cannot speculate on a timeline here. If the winds are right it could happen tomorrow, but the in-development next generation Lean compiler may help reduce the performance gap. There are also practical considerations regarding maintenance in getting the Lean4Lean kernel to be shipped with Lean.

with typing and definitional equality, and a number of theorems and conjectures[3] about the behavior of this relation. These types are a "cleaned-up" version of the corresponding types used in the executable kernel (which uses the locally-nameless variable convention, and includes many conservative extensions such as number and string literals, and primitive projections), to make the presentation of the theory more natural and simplify the task of proving the metatheorems.

- **Program verification.** Given the previous two components, the natural next step is to combine them, such that the executable kernel can have invariants at each stage, culminating in a theorem to the effect that "if `typecheck` e `= ok` τ, then $\vdash [\![e]\!] : [\![\tau]\!]$".[4]

Formalization progress. The project is available at https://github.com/digama0/lean4lean.

Currently, the most complete portion of the project is the **executable typechecker**, which is completely implemented and can typecheck arbitrary Lean projects. The one unsupported Lean kernel feature is the `reduceBool` function, exposed to users via the `native_decide` tactic, which allows the kernel to use the Lean interpreter to run arbitrary Lean code (and in fact, via `@[extern]` annotations, even truly external code such as C or Python) and trust the result. Luckily, this function is (rightfully) viewed with suspicion by most Lean users, and it is banned in many projects such as Mathlib which do not want the expansion of the trusted code base (TCB) implied by this feature.

The **formalized metatheory** portion of the project is well underway, with all of the main definitions stated: expressions, universe levels, contexts, declarations and environments, with the exception of inductive types, which have only the basic scaffolding. (Already many of the "interesting" features of the type theory arise when considering pure MLTT with a universe hierarchy, impredicative `Prop`, and definitional proof irrelevance.) Well-formedness and typing judgments are layered on top of this substrate, with the main player being the $\Gamma \vdash e \equiv e' : \tau$ judgment, defined as an inductive predicate (subsuming both expression typing and definitional equality).

There are some key missing theorems. The most complex theorem in [1] is the unique typing theorem, which says that if $\Gamma \vdash e : \alpha$ and $\Gamma \vdash e : \beta$ then $\Gamma \vdash \alpha \equiv \beta$. It is proved mutually with injectivity of type constructors (e.g. $\mathsf{U}_\ell \equiv \mathsf{U}_{\ell'}$ implies $\ell \equiv \ell'$) and uses a variation on the Tait–Martin-Löf method for proving the Church–Rosser theorem for a specially defined reduction relation which restores confluence while abandoning termination. This theorem is used directly and indirectly in the justification of some of the optimizations employed in the executable kernel.

The final component of the project, **program verification**, is in its early stages. The $\Gamma \vdash [\![e]\!] \Rightarrow e'$ relation which relates the `Expr` type used by the executable kernel to the `VExpr` type used by the theory is defined and some of its basic properties are established. There are similar interpretation relations for most of the other kernel concepts: `LocalContext`, the analogue of the Γ context in the above relations, `Environment` for global declarations, as well as some other material like `NameGenerator` (which is a global counter which ensures new free variables are fresh) and the expression typing cache. But work on giving actual proofs of

[3] The "conjectures" here refer to theorems from [1] with informal proofs which have yet to be formalized. I cannot say with confidence that there are no errors remaining in the informal proof, as it is a very delicate argument. This is part of the reason for wanting to formalize it in the first place.

[4] In actuality the theorem is quite a bit more complicated than this to state, because the typechecker has many additional inputs, such as the environment which supplies the definitions of constants. The interpretation function $[\![\cdot]\!]$ is also not a function, but a relation, because the translation from `Expr` to `VExpr` is not purely syntactic but depends on typing. But this is the essence of the theorem statement.

(the substantially large) kernel functions has yet to properly begin, and concurrent work on a framework for verification of monadic programs by upstream Lean may make this easier.

Future work. The project is quite ambitious and still far from complete, but there are more goals to aspire to beyond this. We already mentioned that `Lean4Lean` does not support `reduceBool`, but this is both possible and desirable for heavy computations which use this feature for performance. It would require formalizing the Lean intermediate representation and the frontend of the compiler, along with some theorems about erasure as in [4]. Furthermore there are more radical changes to kernel reduction (e.g. NbE) which are currently too risky to consider without verification. There are no concrete proposals for changes to reduction at the moment, largely because it's a complete non-starter under the status quo, but it is widely understood that the kernel is very algorithmically inefficient and has performance issues, and this makes it more difficult to practically use implementation techniques such as proof by reflection, widely used in Rocq, in performance-conscious Lean tactics. But there is a large literature on better type checking strategies and Lean4Lean provides a place to experiment and benchmark alternatives, and possibly compensate for the lost 30% performance difference and then some.

References

[1] Mario Carneiro. The Type Theory of Lean. Master's thesis, Carnegie Mellon University, 2019.

[2] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[3] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

[4] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[5] Benjamin Werner. Sets in types, types in sets. In *International Symposium on Theoretical Aspects of Computer Software*, pages 530–546. Springer, 1997.

AdapTT: A Type Theory with Functorial Types

Arthur Adjedj[1,2], Thibaut Benjamin[2], Meven Lennon-Bertrand[2], and Kenji Maillard[3]

[1] ENS Paris-Saclay, Saclay, France
[2] University of Cambridge, Cambridge, United Kingdom
[3] Gallinette Project Team, Inria, Nantes, France

Abstract

How can we make systematic and precise the idea that type formers in dependent type theory are *functorial*? To examine this question, we propose a type theory `AdapTT` with adapters, following ideas of McBride and Nordvall Forsberg [MN21] and Coraglia and Emmenegger [CE23].

Coercions, coercions everywhere Diverse type theories feature operators to turn a term of one type into one of another: coercive subtyping [LA08; LLM24], transport for observational equality [AMS07; PT22], casts for gradual typing [Len+22], or adapters [MN21]. These operators share a similar decomposition of coercions along the structure of the types they operate on. For instance, coercing a function f from a function type $A_0 \to B_0$ to $A_1 \to B_1$ amounts to coercing its argument from A_1 to A_0, and its return value from B_0 to B_1.

Semantically, this shared core can be understood as a form of functoriality of type constructors —see for instance Castellan et al. [CCD17, Lemma 4.8], which is essentially the same as the behaviour of coercions at function types outlined above. However, the statement of this lemma is somewhat contorted, as standard CwFs are not up to the task of expressing this functoriality aspect. Indeed, if we want to talk about functoriality, we need *categories* between which functorial type constructors map. Yet, types in CwFs only form a *set*. As already observed by Coraglia and Emmenegger [CE23], comprehension categories [Jac93] lift this discreteness constraint, providing suitable categorical models of subtyping.

Some pieces are however still missing for a full-fledged understanding of coercions in type theory. First, rather than CwF, Coraglia and Emmenegger [CE23] generalise natural models. Second, the key operation of coercion is not directly explicitated. Finally, only specific type constructors (Π and Σ) are considered, those for which the functorial action is admissible thanks to η-rules. Our goal is to fill in these missing pieces, and provide a good setting to talk about functoriality of type formers and coercions as a general concept.

Adapters As we already hinted at, our first step is to modify the standard CwF structure to $\mathrm{CwF}_{<:}$, which incorporate extra data: types in a $\mathrm{CwF}_{<:}$ form a category rather than a mere set —see full definition of $\mathrm{CwF}_{<:}$ in appendix A. We call arrows between types *adapters*, after McBride and Nordvall Forsberg [MN21]. Each type has an identity adapter, and adapters compose. More importantly, adapters act on terms: if $\Gamma \vdash t : A$ and a is an adapter from A to B, we can form a term $\Gamma \vdash t\,\langle a \rangle : B$, the coercion of t along a. Subtyping can be construed as the special case where types are posets, making the witness adapter irrelevant —only the endpoints matter.

Coercions should satisfy a number of properties in their interactions with identity, composition, substitution... Coraglia and Emmenegger's gCwF structure [CE23] succinctly captures this: although they do not derive it, we have shown that the coercion operation can be constructed in any gCwF. In short, when considered fiberwise over a fixed context, terms form a

discrete opfibration over types. This means their setting is a good semantic fit for adapters, and since they show its equivalence with general comprehension categories, this gives us assurance that $\mathrm{CwF}_{<:}$, which reformulate gCwF in the language of CwF, are a reasonable notion.

Parameter contexts We now have an idea of what the codomain of a functorial type constructor should be. But what should its domain be? We propose to use *parameter contexts*, which contain two kinds of variable. The usual, term-level ones, and *type variables*. These come with two pieces of information: a variance, which we will use to differentiate between co- and contravariant functors[1]; and a context of bound variable that a type can depend on. As an example, the following represents the parameters of the dependent function type constructor Π, with one contravariant type, and a covariant one depending on the first

$$\Gamma_\Pi \coloneqq (X : \mathrm{Ty}_-) \triangleright (Y : (x : X).\, \mathrm{Ty}_+)$$

A substitution into that context $\Delta \vdash \sigma : \Gamma_\Pi$ accordingly consists of two types, the second depending on the first. Since we have arrows between types, we also get arrows between these substitutions: if $\Delta \vdash \tau : \Gamma_\Pi$ is another substitution into Γ_Π, a *transformation* $\Delta \vdash \mu : \sigma \Rightarrow \tau$ consists of an adapter between $\tau(X)$ and $\sigma(X)$ (since X is contravariant), and one between $\sigma(Y)$ and $\tau(Y)$, suitably lying over the first. This is exactly the data we need to build an adapter between $\Pi\, x : \sigma(X).\sigma(Y)$ and $\Pi\, x : \tau(X).\tau(Y)$ by pre- and post-composition, as above.

Generalising this example, a type constructor of "arity" Γ amounts to a type $\Gamma \vdash T$, which generates types in all other contexts and the relevant adapters, by the respective actions of substitutions and transformations. In particular, such a type gives rise to a functor from the category of substitutions $\mathrm{Sub}(\Delta, \Gamma)$ to that of types in Δ: given two substitutions $\Delta \vdash \sigma, \tau : \Gamma$ and a transformation $\Delta \vdash \mu : \sigma \Rightarrow \tau$, we get an adapter $T[\![\mu]\!]$ between $T[\sigma]$ and $T[\tau]$.

We succinctly describe this with a 2-category of parameter contexts, substitutions and transformations, the latter built of adapters. The above intuition can then be quickly summarised by saying that Ty is a 2-functor. In the most general view, a type constructor can be understood as a natural transformation from a (Cat-valued) presheaf on the category of contexts to this 2-functor Ty. Our notion of parameter contexts is rich enough to make many such presheaves representable. By the (2-categorical) Yoneda lemma —which gives an isomorphism between the categories $\mathrm{Cat}^{\mathrm{Ctx}^{\mathrm{op}}}[\mathrm{y}\,\Gamma, \mathrm{Ty}]$ and $\mathrm{Ty}(\Gamma)$— parameter contexts let us represent these type constructors and their functorial action more directly, as we did with Γ_Π.

We are exploring the formalisation of all this in Agda as a quotient-inductive-inductive-recursive type, representing the syntax of a putative type theory `AdapTT`. In particular, we can check that type formers such as Π, Σ, or the identity type, adapt to this functorial setting. And they do, following the familiar pattern of structural coercions/casts/transport.

The next direction we hope to explore, which was part of our original motivation, is to provide a setting to generally derive this functorial structure for datatypes. That is, to design a theory of signatures which incorporates —and checks— variance information, and generically derives the adapter equivalent of a map operation [LLM24].

Towards 2-CwF The 2-categorical aspects of the category of contexts suggests a connection to the 2-dimensional type theory 2DTT of Licata and Harper [LH11]. Indeed, internalizing type variables as quantifications over a universe of sets yields a theory very close to 2DTT. However, 2DTT does not have an explicit judgement for the categorical structure on types corresponding to adapters, merely relying on the corresponding notion between terms. A unifying notion of 2-dimensional type theory should combine our $\mathrm{CwF}_{<:}$ with type variables and 2DTT.

[1] For simplification we do not consider equivariance, which should however be relatively easy to add.

A Full definition of CwF with adapters

Definition 1 (Families indexed by categories)**.** $\mathbf{Fam}_{<:}$ is the category whose objects are pairs of a category X and a functor $X \to \mathbf{Set}$. An arrow $(X, F) \to (Y, G)$ is a pair of a functor $H : X \to Y$ and a natural transformation $F \dot{\to} G \circ H$. Abstractly (and ignoring size issues), $\mathbf{Fam}_{<:}$ is the lax slice category $\mathbf{Cat} /\!\!/ \mathbf{Set}$.

Definition 2 (Categories with families and adapters ($\mathrm{CwF}_{<:}$))**.** A *category with families and adapters* ($\mathrm{CwF}_{<:}$) is given by the following data.

- A category Ctx whose objects are called *contexts* and arrows *substitutions*.
- A functor $T : \mathrm{Ctx}^{\mathrm{op}} \to \mathbf{Fam}_{<:}$, that is, given a context $\Gamma : \mathrm{Ctx}$, we have
  - a category $\mathrm{Ty}(\Gamma)$ of *types* in Γ, we write $\mathrm{Ad}(\Gamma, A, B)$ (*adapters*) for the collection of arrows between two types A and B;
  - for any $A : \mathrm{Ty}(\Gamma)$, a set of *terms* $\mathrm{Tm}(\Gamma, A)$;
  - an action of substitution $\cdot\,[\cdot]$ on types, adapters and terms,
  - and an action $\cdot\,\langle\cdot\rangle$ of adapters on terms: if $t : \mathrm{Tm}(\Gamma, A)$ and $a : \mathrm{Ad}(\Gamma, A, B)$, then $t\,\langle a\rangle : \mathrm{Tm}(\Gamma, B)$;
  - suitable equalities for the compatibility of the actions of substitution and adapters with identities, compositions and each other.
- For any context Γ and type $A : \mathrm{Ty}(\Gamma)$, a *context extension* $\Gamma \triangleright A$, equipped with
  - a substitution $\mathrm{wk} : \Gamma \triangleright A \to \Gamma$ (*weakening*)
  - and a term $\mathrm{vz} : \mathrm{Tm}(\Gamma \triangleright A, A\,[\mathrm{wk}])$ (*variable zero*)
  - in a universal way: for any context Δ equipped with $\sigma : \Delta \to \Gamma$ and $t : \mathrm{Tm}(\Delta, A\,[\sigma])$, there exists a substitution $\sigma \triangleright t : \Delta \to \Gamma \triangleright A$ (*substitution extension*) such that $\mathrm{wk} \circ (\sigma \triangleright t) = \sigma$ and $\mathrm{vz}\,[\sigma \triangleright t] = t$, which is unique, *i.e.* for any $\sigma : \Delta \to (\Gamma \triangleright A)$ we have $\sigma = (\mathrm{wk} \circ \sigma) \triangleright \mathrm{vz}\,[\sigma]$.

Theorem 3. *Every $\mathrm{CwF}_{<:}$ in the sense of definition 2 is a gCwF in the sense of Coraglia and Emmenegger [CE23] that is moreover split, and vice-versa any split gCwF is a $\mathrm{CwF}_{<:}$.*

This notion of category of families with adapters can alternatively be presentated via the following Second-Order Generalized Algebraic Theory [Uem21; KX24]:

$$\mathrm{ty} \;:\; \mathbf{Sort} \qquad \mathrm{ad} \;:\; \mathrm{ty} \to \mathrm{ty} \to \mathbf{Sort}$$
$$\mathrm{tm} \;:\; \mathrm{ty} \to \mathbf{RepSort} \qquad \cdot\,\langle\cdot\rangle \;:\; \{A\,B : \mathrm{ty}\} \to \mathrm{ad}\,A\,B \to \mathrm{tm}\,A \to \mathrm{tm}\,B$$
$$\mathrm{id} \;:\; \{A : \mathrm{ty}\} \to \mathrm{ad}\,A\,A \qquad \circ \;:\; \{A\,B\,C : \mathrm{ty}\} \to \mathrm{ad}\,B\,C \to \mathrm{ad}\,A\,B \to \mathrm{ad}\,A\,C$$
$$\mathrm{idl} \;:\; \{A\,B : \mathrm{ty}\}(a : \mathrm{ad}\,A\,B) \to \mathrm{id} \circ a \equiv a \qquad \mathrm{idr} \;:\; \{A\,B : \mathrm{ty}\}(a : \mathrm{ad}\,A\,B) \to a \circ \mathrm{id} \equiv a$$
$$\mathrm{assoc} \;:\; \{A\,B\,C\,D : \mathrm{ty}\}(a : \mathrm{ad}\,A\,B)(b : \mathrm{ad}\,B\,C)(c : \mathrm{ad}\,C\,D) \to c \circ (b \circ a) \equiv (c \circ b) \circ a$$
$$\mathrm{adapt/id} \;:\; \{A : \mathrm{ty}\}(t : \mathrm{tm}\,A) \to t\langle \mathrm{id}\rangle \equiv t$$
$$\mathrm{adapt}/\circ \;:\; \{A\,B\,C : \mathrm{ty}\}(a : \mathrm{ad}\,A\,B)(b : \mathrm{ad}\,B\,C)(t : \mathrm{tm}\,A) \to t\langle b \circ a\rangle \equiv t\langle a\rangle\langle b\rangle$$

This presentation exhibits the adapters as the hom-sets of a category structure on types (internally to presheaves over the category of contexts) and their action on terms as an (internal) presheaf structure over the category of types and adapters.

References

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "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.

[CCD17] Simon Castellan, Pierre Clairambault, and Peter Dybjer. "Undecidability of equality in the free locally cartesian closed category (extended version)". In: *Logical Methods in Computer Science* 13 (2017).

[CE23] Greta Coraglia and Jacopo Emmenegger. *Categorical models of subtyping*. 2023. DOI: 10.48550/arxiv.2312.14600. arXiv: 2312.14600 [cs.LO].

[Jac93] Bart Jacobs. "Comprehension Categories and the Semantics of Type Dependency". In: *Theoretical Computer Science* 107.2 (1993), pp. 169–207. DOI: 10.1016/0304-3975(93)90169-T.

[KX24] Ambrus Kaposi and Szumi Xie. "Second-Order Generalised Algebraic Theories: Signatures and First-Order Semantics". In: *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*. Ed. by Jakob Rehof. Vol. 299. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 10:1–10:24. ISBN: 978-3-95977-323-2. DOI: 10.4230/LIPICS.FSCD.2024.10. URL: https://doi.org/10.4230/LIPIcs.FSCD.2024.10.

[LA08] Zhaohui Luo and Robin Adams. "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. URL: https://www.cambridge.org/core/article/structural-subtyping-for-inductive-types-with-functorial-equality-rules/03A1AD06A0D2F0E6037C2D54DBA68CCC.

[Len+22] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. "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.

[LH11] Daniel R. Licata and Robert Harper. "2-Dimensional Directed Type Theory". In: *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011*. Ed. by Michael W. Mislove and Joël Ouaknine. Vol. 276. Electronic Notes in Theoretical Computer Science. Elsevier, 2011, pp. 263–289. DOI: 10.1016/J.ENTCS.2011.09.026.

[LLM24] Théo Laurent, Meven Lennon-Bertrand, and Kenji Maillard. "Definitional Functoriality for Dependent (Sub)Types". In: *33rd European Symposium on Programming, ESOP 2024*. Ed. by Stephanie Weirich. Vol. 14576. Lecture Notes in Computer Science. Springer, 2024, pp. 302–331. DOI: 10.1007/978-3-031-57262-3_13.

[MN21] Conor McBride and Frederik Nordvall Forsberg. "Functorial Adapters". In: *27th International Conference on Types for Proofs and Programs*. 2021.

[PT22] Loïc Pujet and Nicolas Tabareau. "Observational Equality: Now for Good". In: *Proc. ACM Program. Lang.* 6.POPL (2022). DOI: 10.1145/3498693.

[Uem21] Taichi Uemura. "Abstract and Concrete Type Theories". PhD thesis. University of Amsterdam, 2021. URL: https://eprints.illc.uva.nl/id/document/12150.

Cohomology in Synthetic Stone Duality

Felix Cherubini, Thierry Coquand, Freek Geerligs, and Hugo Moeneclaey *

University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden

Peter Scholze and Dustin Clausen [CS24] introduced light condensed sets, defined as sheaves on the site of light profinite sets. They can be used as an alternative to topological spaces. Synthetic Stone duality is an extension of homotopy type theory by four axioms, which was introduced in [Che+24]. In this article, it was proven that $H^1(S, \mathbb{Z}) = 0$ for S a Stone space, that $H^1(X, \mathbb{Z})$ for X compact Hausdorff can be computed using Čech cohomology and that $H^1(\mathbb{I}, \mathbb{Z}) = 0$ where $\mathbb{I}$ is the unit interval. In this talk we will present the extension of these results to higher cohomology groups with non-constant countably presented abelian groups as coefficients. Those are synthetic analogues of results from Roy Dyckhoff [Dyc76a; Dyc76b].

Synthetic Stone Duality In our setting, Stone spaces are precisely *countable* sequential limits of finite sets, making them analogous to *light* profinite sets. The construction of a model making this analogy rigorous is still work in progress.

The axioms of synthetic Stone duality postulate Stone duality (Stone spaces are equivalent to countably presented Boolean algebras), completeness (non-empty Stone spaces are merely inhabited), dependent choice and a *local-choice* axiom. The latter says that given a Stone space S and a type family B over S such that $\Pi_{x:S}\|B(x)\|$, there merely is a Stone space T and a surjection $s : T \twoheadrightarrow S$ such that $\Pi_{x:T}B(s(x))$. Local choice is crucial when performing the cohomological computations mentioned below.

Many traditional properties of Stone spaces can be shown synthetically, sometimes phrased in a more type-theoretic way, e.g. Stone spaces are closed under Σ-types. Open and closed propositions can be defined, inducing a topology on any type such that any map is continuous. This topology is as expected for Stone spaces and compact Hausdorff spaces (i.e. quotient of Stone spaces by closed equivalence relations).

One important example of compact Hausdorff space is the real interval $\mathbb{I}$, from which the type $\mathbb{R}$ of real numbers is constructed. This is equivalent to the usual constructions of both Cauchy and Dedekind reals. As in [Shu18], it is important to distinguish topological spaces like $\mathbb{S}^1 := \{x, y : \mathbb{R} \mid x^2 + y^2 = 1\}$ from homotopical spaces like the higher inductive 1-type S^1.

Despite topological spaces being sets, they can have non-trivial cohomology. Indeed, for any type X and dependent abelian group $A : X \to \mathrm{Ab}$, we use the usual synthetic definition of the n-th cohomology group $H^n(X, A)$ as $\|\Pi_{x:X}K(A_x, n)\|_0$ where $K(A_x, n)$ is the n-th Eilenberg Mac-Lane space with coefficient A_x. In [Che+24], it is proven that $H^1(\mathbb{S}^1, \mathbb{Z}) = \mathbb{Z}$, despite $\mathbb{S}^1$ being a set.

We prove Barton and Commelin's condensed type theory axioms [Bar24] in synthetic Stone duality, as well as dependent generalisations of them. These are used to show that any compact Hausdorff space X interact well with any family of countably presented abelian groups $A : X \to \mathrm{Ab}_{\mathrm{cp}}$.

Vanishing of higher cohomology of Stone spaces First we prove that $H^1(S, A) = 0$ for all S Stone and $A : S \to \mathrm{Ab}_{\mathrm{cp}}$. We assume $\alpha : \prod_{x:S} K(A_x, 1)$, by local choice we get a surjection $p : T \twoheadrightarrow S$ with T Stone which trivialises α. Then we get an approximation of p as a sequential limit of surjective maps $p_k : T_k \to S_k$ between finite sets, we check that the induced trivialisation

*Speaker.

over T_k gives a trivialisation over S_k, and conclude through our dependent generalisations of Barton and Commelin's axioms that α is trivial over $\lim_k S_k = S$.

We follow an idea due to David Wärn [BCW23, Theorem 3.4] to go from $H^1(S, A) = 0$ to $H^n(S, A) = 0$ for all $n > 0$. The key idea is to proceed by induction on $n > 0$, generalising the induction hypothesis from $H^n(S, A) = 0$ to:

(i) $K(\prod_{x:S} A_x, n) \to \prod_{x:S} K(A_x, n)$ is an equivalence, directly implying $H^n(S, A) = 0$.

(ii) $K(\prod_{x:S} A_x, n+1) \to \prod_{x:S} K(A_x, n+1)$ is an embedding.

Assume (i) and (ii) for $n > 0$, let's prove (i) and (ii) for $n + 1$. (ii) follows immediately from (i). To prove (i), by induction hypothesis (ii) it is enough to prove that $\prod_{x:S} K(A_x, n+1)$ is connected, i.e. $H^{n+1}(S, A) = 0$. We assume $\alpha : \prod_{x:S} K(A_x, n+1)$, by local choice we get a trivialisation $p : T \twoheadrightarrow S$ of α with T Stone. Denoting by T_x the fiber of p over x, we consider the exact sequence $0 \to A_x \to A_x^{T_x} \to L_x \to 0$ giving an exact sequence:

$$H^n(S, L) \to H^{n+1}(S, A) \to H^{n+1}(S, \lambda x.A_x^{T_x})$$

By induction hypothesis (i) we have that $H^n(S, L) = 0$ so we have an injection:

$$H^{n+1}(S, A) \to H^{n+1}(S, \lambda x.A_x^{T_x})$$

By induction hypothesis (ii) the map:

$$H^{n+1}(S, \lambda x.A_x^{T_x}) \to H^{n+1}(\Sigma_{x:S} T_x, A_x) = H^{n+1}(T, A \circ p)$$

is an injection so that we get an injection:

$$p^* : H^{n+1}(S, A) \to H^{n+1}(T, A \circ p)$$

But p trivialises α so $p^*(\alpha) = 0$, therefore $\alpha = 0$.

Čech cohomology for compact Hausdorff spaces Given a compact Hausdorff space X, a Čech cover for X consists of a Stone space S and a surjective map $p : S \twoheadrightarrow X$. By definition any compact Hausdorff space has a Čech cover.

Given such a Čech cover and $A : X \to \mathrm{Ab}_{\mathrm{cp}}$, we define its Čech complex as:

$$\Pi_{x:X} A_x^{S_x} \to \Pi_{x:X} A_x^{S_x \times S_x} \to \cdots$$

with the boundary maps defined as expected, and its Čech cohomology $\check{H}^k(X, A)$ as the k-th homology group of its Čech complex. It is clear that $H^0(X, A) = \check{H}^0(X, A)$.

From hypothesis (i) in the previous paragraph, for $n > 0$ we get an exact sequences:

$$H^{n-1}(X, \lambda x.A_x^{S_x}) \to H^{n-1}(X, L) \to H^n(X, A) \to 0$$

By direct computations, for $n > 0$ we get an exact sequence:

$$\check{H}^{n-1}(X, \lambda x.A_x^{S_x}) \to \check{H}^{n-1}(X, L) \to \check{H}^n(X, A) \to 0$$

We conclude by induction on n that $H^n(X, A) = \check{H}^n(X, A)$ for all n, so that in particular Čech cohomology does not depend on the Čech cover. For this induction to go through it is crucial that $A_x^{S_x}$ is countably presented, which follows from Barton and Commelin's axioms. Using this result and finite approximations of a well-chosen Čech cover of $\mathbb{I}$, we can check that $H^n(\mathbb{I}, A) = 0$ for all $n > 0$ and $A : \mathbb{I} \to \mathrm{Ab}_{\mathrm{cp}}$.

References

[Bar24] Reid Barton. *Directed aspects of condensed type theory*. 2024. URL: https://www.math.uwo.ca/faculty/kapulkin/seminars/hottestfiles/Barton-2024-09-26-HoTTEST.pdf (cit. on p. 1).

[BCW23] Ingo Blechschmidt, Felix Cherubini, and David Wärn. *Čech Cohomology in Homotopy Type Theory*. 2023. URL: https://www.felix-cherubini.de/cech.pdf (cit. on p. 2).

[Che+24] Felix Cherubini et al. "A Foundation for Synthetic Stone Duality". In: (2024). arXiv: 2412.03203 [math.LO]. URL: https://arxiv.org/abs/2412.03203 (cit. on p. 1).

[CS24] Dustin Clausen and Peter Scholze. *Analytic Stacks*. Lecture series. 2023-2024. URL: https://www.youtube.com/playlist?list=PLx5f8IelFRgGmu6gmL-Kf_Rl_6Mm7juZO (cit. on p. 1).

[Dyc76a] Roy Dyckhoff. *Categorical methods in dimension theory*. English. Categor. Topol., Proc. Conf. Mannheim 1975, Lect. Notes Math. 540, 220-242 (1976). 1976 (cit. on p. 1).

[Dyc76b] Roy Dyckhoff. "Projective resolutions of topological spaces". English. In: *J. Pure Appl. Algebra* 7 (1976), pp. 115–119. ISSN: 0022-4049. DOI: 10.1016/0022-4049(76)90069-4 (cit. on p. 1).

[Shu18] Michael Shulman. "Brouwer's fixed-point theorem in real-cohesive homotopy type theory". In: *Mathematical Structures in Computer Science* 28.6 (2018), pp. 856–941. DOI: 10.1017/S0960129517000147 (cit. on p. 1).

Representing type theories in two-level type theory

Nicolai Kraus and Tom de Jong

University of Nottingham, Nottingham, UK
{nicolai.kraus,tom.dejong}@nottingham.ac.uk

Summary. We describe how two-level type theory can be used to represent a type theory by turning structural extensions into axiomatic extensions, using Riehl and Shulman's simplicial type theory as an example. This talk explains the motivation behind work in progress.

2LTT and extensions of type theories. *Two-level type theory (*2LTT*)* [18, 2, 3, 1] is a framework that allows internalising some of the meta-theory of a given type theory. Simplified, it can be described as one type theory sitting inside another type theory (*inner/fibrant* and *outer/strict/exo-level*). A useful example arises if we start with a version of extensional MLTT (as the strict layer) and assume that some types carry a flag marking them as fibrant, done in a way that ensures that the fibrant layer is exactly homotopy type theory (HoTT) [17]. This situation is modelled by some presheaf models such as the simplicial sets model, where only some maps (Kan fibrations) correspond to (fibrant) types [9]. With the correct specification, the fibrant layer is exactly as expressive as HoTT in the sense of a conservativity property [2, 10].

In the current work, we explore an application of 2LTT in the context of extensions of type theories. We consider two types of extensions:

1. *Axiomatic extensions*, where a type theory T_2 can be represented as a type theory T_1 extended with axioms. Examples are the extension MLTT $\hookrightarrow$ MLTT + funext, or MLTT $\hookrightarrow$ HoTT, if HoTT is the theory developed in the book [17] (where univalence is added as an axiom), without the judgemental computational rules for higher inductive types.
2. *Structural extensions*, where T_2 has all (or most) of the structure of T_1 but also contains additional structural rules. Examples are the extensions of MLTT to (certain versions of) Cubical Type Theory [4], or of HoTT to Riehl and Shulman's type theory of $(\infty, 1)$-categories [15].

Generally speaking, axiomatic extensions are the nicer ones. If we are already familiar with T_1, we merely need to get intuition for internal postulates in order to understand what we might be able to prove in T_2, and if we have a proof assistant for T_1, we can use it as a proof assistant for T_2 by simply adding a postulate. The same is not true for structural extensions; for example, cubical Agda depends on modifications to Agda's source code by the development team, and for Riehl and Shulman's type theory, Kudasov developed the new proof assistant Rzk [11, 12].

The extension HoTT $\hookrightarrow$ 2LTT is a structural extension but it is, in some sense, harmless; since one is ultimately only interested in the fibrant fragment, the mentioned conservativity guarantees that the same internal theorems hold as in HoTT. The point is that:

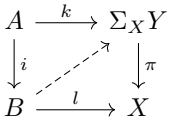
A given structural extension of HoTT *might be an axiomatic extension of* 2LTT*.*

While this seems not necessarily useful from a computational point of view, the axiomatic extension may be easier to get intuition for, and to reason about, than the structural extension; and the setup may enable us to view structurally different type theories in the same setting, a situation somewhat reminiscent of a logical framework [8]. Moreover, the additional language that 2LTT provides may enable us to study properties in the type theory that would ordinarily be meta-theoretic. As a more practical benefit, using Agda's `--two-level` flag [5], we might be able to directly use Agda to implement the new type theory, bypassing the need for the development of a custom-made proof assistant.

Simplicial type theory via two-level type theory. In the remainder of this talk abstract, we sketch how Riehl and Shulman's type theory for $(\infty, 1)$-categories [15] (*simplicial type theory*, STT) can be represented within 2LTT. The theory STT can be described as HoTT with two additional ingredients. The first are two additional context layers that make it possible to talk about *shape inclusions* and *extension types*; the second is the assumption of a directed interval on the context level which ensures that the usual simplicial shapes, such as horns or boundary inclusions, can be constructed as shape inclusions.

Extension types are a concept that multiple type theories make use of, cf. [19] for a summary. The idea is to allow a version of dependent functions $f : \Pi_X Y$, where the type fixes the value of f along some (possibly meta-theoretic) map $i : A \to X$. For example in the cubical case [4], one considers extensions of functions defined on partial boxes.

In the setting of 2LTT, the concept of extension types is very natural. A function $i : A \to B$ on the strict layer is called a *cofibration* (corresponding to what some type theories call shape inclusions) if the *Leibniz exponential* [16] (a.k.a. *pullback exponential*) of i preserves fibrations and trivial fibrations [2, §3.4]. This condition means that, given a fibrant family Y over a not-necessarily fibrant X in a strictly commuting square as shown on the right, the type of strict fillers (i.e. the type of functions $B \to \Sigma_X Y$) is fibrant, and fibrewise contractible if Y is a contractible family. The extension type described above occurs when l is the identity on X.



The second assumption of STT is an interval on the context level, equipped with a bounded linear order, and the new structural rules make it possible to create standard simplicial maps from this interval. This can be replicated directly in 2LTT, with the interval on the strict layer satisfying a cofibrancy assumption. An alternative way is to postulate a functor shape from the category $\mathcal{S}$ to the universe of strict types, where $\mathcal{S}$ is the subcategory of simplicial sets spanned by subfunctors of representables, with the assumption that monos are mapped to cofibrations and finite (co)limits are preserved. The original interval with order can be used to define such a functor and is used in STT to construct the components that the functor provides, which leads us to believe that postulating the functor directly may be the easier approach.

In 2LTT, the pairs (cofibrations, trivial fibrations) and (trivial cofibrations, fibrations) are *orthogonal factorisation systems*, and STT is concerned with an orthogonal factorisation system that lies between these, generated by the cofibration $\Lambda^2_1 \hookrightarrow \Delta^2$ as a member of the left class. If $X \to 1$ lies in the right class, then X is called a *Segal type*; if it additionally satisfies a univalence/completeness condition, it carries the structure required for it to be considered an $(\infty, 1)$-category, following Rezk's model of complete Segal spaces [14].

One appeal of representing STT within 2LTT is that the two layers of the latter can be used to cleanly separate the "combinatorial/set level part" from the "homotopical part" of STT. The first is concerned with general results about simplicial sets, e.g. Joyal's lemma (cf. [13, Lem. 2.3.2.1]), and works on the strict/outer level; the second is the part of the theory which depends on working with spaces rather than sets, here given by the inner/fibrant theory.

As demonstrated by recent work [6, 7], STT can also be represented as an axiomatic extension of HoTT. This requires the stronger assumption that the interval is fibrant, so that everything above can be carried out in HoTT. An explanation for why this works well is that, assuming the corners of the square above are fibrant, the type of strict fillers (i.e. the "meta theoretic" extension type) is equivalent to the type of homotopy fillers (the "internal" extension type). The assumption that the interval is fibrant is easily justified but not completely free: a model that gets lost is the original simplicial sets model [9], where the interval is cofibrant but not fibrant. However, while it can be viewed as a model of STT with the 2LTT approach, the model is "too small" to have undirected equalities, and all types turn out to be ∞-groupoids.

References

[1] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending homotopy type theory with strict equality. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leipniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2018.

[2] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *Mathematical Structures in Computer Science*, 33(8):688–743, 2023.

[3] Paolo Capriotti. *Models of type theory with strict equality*. PhD thesis, University of Nottingham, 2017. https://eprints.nottingham.ac.uk/id/eprint/39382.

[4] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leipniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2018.

[5] Agda documentation. Two-level type theory. https://agda.readthedocs.io/en/v2.6.3/language/two-level.html.

[6] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. Directed univalence in simplicial homotopy type theory. arXiv: 2407.09146, 2024.

[7] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. The Yoneda embedding in simplicial homotopy type theory. arXiv: 2501.13229, to appear in the proceedings of *Logic in Computer Science 2025*, 2025.

[8] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[9] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after Voevodsky). *Journal of the European Mathematical Society*, 23(6):2071–2126, 2021.

[10] András Kovács. Staged compilation with two-level type theory. In Philip Wadler, editor, *ICFP*, volume 6 of *Proceedings of the ACM on Programming Languages*, pages 540–569. Association for Computing Machinery, 2022.

[11] Nikolai Kudasov. Rzk proof assistant. https://github.com/rzk-lang/rzk.

[12] Nikolai Kudasov, Emily Riehl, and Jonathan Weinberger. Formalizing the ∞-categorical Yoneda lemma. In Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy, editors, *CPP '24*, pages 274–290. Association for Computing Machinery, 2024.

[13] Jacob Lurie. *Higher Topos Theory*, volume 170 of *Annals of Mathematics Studies*. Princeton University Press, 2009.

[14] Charles Rezk. A model for the homotopy theory of homotopy theory. *Transactions of the American Mathematical Society*, 353(3):973–1007, 2001.

[15] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories. *Higher Structures*, 1(1):147–224, 2017.

[16] Emily Riehl and Dominic Verity. The theory and practice of Reedy categories. *Theory and Applications of Categories*, 29(9):256–301, 2013.

[17] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[18] Vladimir Voevodsky. A simple type theory with two identity types. Unpublished note, available at https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf, 2013.

[19] Tesla Zhang. Three non-cubical applications of extension types. arXiv: 2311.05658, 2023.

Extensional concepts in intensional type theory, revisited

Krzysztof Kapulkin[1] and Yufeng Li[2]

[1] University of Western Ontario, London, Ontario, Canada
kkapulki@uwo.ca
[2] University of Cambridge, Cambridge, United Kingdom
yufeng.li@cl.cam.ac.uk

In his Ph.D. dissertation, Hofmann [Hof95; Hof97] constructs an interpretation of extensional type theory in intensional type theory, subsequently proving a *conservativity* result of the former over the latter extended by the principles of functional extensionality and of uniqueness of identity proofs (UIP), cf. [Hof95, §3.2]. Interestingly, Hofmann's proof is "stronger" than the statement of his theorem, as the requisite language in which to speak of conservativity and equivalence of dependent type theories did not exist at the time.

A major insight came as a result of the development of homotopy type theory. In particular, Voevodsky's definition, ca. 2009, of when a morphism in a model of dependent type theory is a *weak equivalence* allows one to consider different homotopy-theoretic structures both within such models and the categories thereof.

In [KL18], it was observed that the category of models of a dependent type theory carries the structure of a left semi-model category. This structure was subsequently used by Isaev [Isa18] to define a *Morita equivalence* of dependent type theories. In essence, two theories are Morita equivalent if their categories of models are equivalent in a suitable "up-to-homotopy" sense. More precisely, a Morita equivalence between theories is a translation between them that induces a Quillen equivalence (the correct notion of equivalence for left semi-model structures) between their left semi-model categories of models. Perhaps unsurprisingly, Isaev cites Hofmann's theorem as one of the motivating examples behind his definition, without actually proving it to be one.

In our recent publication [KL25], we give a direct proof of Morita equivalence between the extensional type theory and the intensional type theory extended by the principles of functional extensionality and of uniqueness of identity proofs. While Hofmann proves that the initial models of these theories are suitably equivalent, we generalize this result to all possible extensions of the base theories by types and terms, including propositional equalities. In homotopy-theoretic terms, these are exactly the *cofibrant* extentions.

Therefore, thanks to proving Morita equivalence, one does not need to prove an analogue of Hofmann's result for any new extension but instead appeal to our result addressing all extensions once and for all. As new variants and extensions of intensional type theory are constantly proposed, this reduces the burden of proving their expected properties by making what should be formal formal.

Our proof follows Hofmann's quite closely but requires a major innovation to account for all possible (cofibrant) extensions. In [Hof97], Hofmann describes a class of context morphisms termed *propositional isomorphisms* by inspecting the outermost type former in (the last type of) each context and collapses these maps to identities, thus obtaining a functor from the syntactic model of intensional type theory to that of extensional type theory.

Our general approach is nearly identical: first, we describe the *extensional kernel* of a cofibrant model of intensional type theory, which we then collapse in a construction reminiscent of Hofmann's to obtain a (cofibrant) model of extensional type theory. This explicit construction then allows one to directly verify that it produces a model of extensional type theory and yields Morita equivalence between the theories. Because of working with syntactic categories,

Hofmann is able to inspect the outermost constructor in each context. This property need not hold in every model, but it does hold in the cofibrant ones, which is therefore sufficient to follow the remainder of Hofmann's strategy to establish a Morita equivalence.

In this talk, after reviewing the necessary background, we give an overview of the proof, including a detailed comparison with Hofmann's approach.

References

[Hof95] Martin Hofmann. "Extensional concepts in intensional type theory". PhD thesis. University of Edinburgh, 1995.

[Hof97] Martin Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS Distinguished Dissertations. Springer-Verlag London, Ltd., London, 1997, pp. xii+214.

[Isa18] Valery Isaev. "Morita equivalences between algebraic dependent type theories". preprint. 2018. arXiv: 1804.05045.

[KL18] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. "The homotopy theory of type theories". In: *Advances in Mathematics* 337 (2018).

[KL25] Krzysztof Kapulkin and Yufeng Li. "Extensional concepts in intensional type theory, revisited". In: *Theoretical Computer Science* 1029 (2025).

Lightweight Agda Formalization of Denotational Semantics

Peter D. Mosses[1][2]

[1] Delft University of Technology, The Netherlands
p.d.mosses@tudelft.nl
[2] Swansea University, United Kingdom

This paper introduces a lightweight approach to formalization of Scott–Strachey style denotational semantics in Agda. In contrast to previous approaches, it allows definitions of denotations in λ-notation to be embedded straightforwardly in Agda without significant changes. A lightweight Agda formalization of the standard denotational semantics of the untyped λ-calculus is given to illustrate the simplicity of the approach; lightweight Agda formalizations of denotational definitions of PCF and core Scheme are available online [11].

The lightweight approach presented here simply assumes that *all Agda types are Scott-domains, and that all Agda functions have fixed points.* Such assumptions are obviously unsound, but the Agda proof assistant accepts them, and their unsoundness does not affect checking definitions for well-formedness.

Motivation. The author's original motivation for formalizing denotational semantics was to validate a semantics of inheritance [10]. The aim was to check the soundness not only of the results stated in [2], but also of the individual proof-steps, and the Agda proof assistant seemed particularly well-suited for that. The resulting formalization [9] is reasonably lightweight. Type-checking it with Agda revealed several subtle flaws in the original denotational definition; after they had been fixed, Agda successfully checked the proof-steps of various lemmas.

Scott domain theory. In conventional Scott–Strachey denotational semantics [20, 21, 22, 27, 29], the denotation of a program phrase is an element of a Scott-domain: an ω-complete poset with a least element, possibly with further properties. Flat domains are defined as lifted sets, and domains are closed under domain constructors including lifting, cartesian product, and separated sum. Domains can also be defined recursively (up to isomorphism) without restrictions. Functions on domains defined in λ-notation are continuous, ensuring that endo-functions have least fixed points. More recent presentations of Scott domain theory [1, e.g.] define also predomains, which need not have a least element, and thus include ordinary sets (discretely ordered) as well as domains.

Agda formalization of Scott domain theory. The Agda *TypeTopology* library is based on univalent foundations. It includes modules for Scott domain theory, and illustrates their use in denotational definitions of PCF and the untyped λ-calculus [5].

This Agda formalization of domain theory corresponds directly to the usual set-theoretic definitions: a domain consists of a carrier type together with a partial order relation, its least element, and proofs of the required completeness properties; a continuous function between domains is an underlying function between their carrier types, paired with a proof of its continuity. Similarly for predomains.

Currently, the formalization requires definitions of denotations in λ-notation to include explicit continuity proofs, and subsequently discard the proof terms when applying functions. This prevents direct embedding of λ-notation from conventional denotational definitions, and seems quite impractical for formalizing the denotational semantics of larger languages (especially in continuation-passing style, e.g., for Scheme [19]).

Agda formalization of synthetic domain theory. Instead of formalizing the standard set-theoretic definitions of domains and continuous functions, synthetic domain theory (SDT) axiomatizes domains as kinds of sets in intuitionistic set-theory. Endofunctions on such sets have fixed points, and recursive set equations have solutions. SDT was suggested by Dana Scott [23], about 10 years after his initial development of domain theory.

Most of the published theoretical work on SDT [3, 7, 12, 13, 16, 18, 24, 25, 28] concentrated mainly on sorting out the underlying mathematical framework of what properties domains have, and on studying models of such domains. However, Bernhard Reus [14] also formalized SDT in the *Lego* proof assistant. The formalization relies on impredicativity and proof-irrelevance [15], which prevents porting it straightforwardly to Agda.

Alex Simpson's development of SDT [24] is based on intuitionistic ZF set theory. The generality of the approach is illustrated by a denotational semantics of FPC, a recursively-typed λ-calculus with sum and product types. In op. cit. (§3) he wrote: "it seems likely that, with appropriate reformulations, the development of this paper could be carried out in the (predicative) context of Martin-Löf's Type Theory", but apparently its formalization in Agda has not yet been attempted.

It appears that the only formalizations of SDT so far developed in Agda are based on guarded domain definitions in clocked cubical type theory [4, 6, 26]. However, denotations then involve step-indexing, so they are generally more intensional than in conventional Scott domain theory.

Lightweight Agda formalization. The Agda code presented below is a lightweight formalization of a standard denotational semantics of the untyped call-by-name λ-calculus, following [17, §10.5]. The complete source code is available online [11].

Abstract syntax. Denotational semantics conventionally defines the abstract syntax of a language by a context-free grammar. Agda doesn't include grammars, but it is quite straightforward to transform a grammar to inductive datatype definitions with the same interpretation. The following datatype uses ordinary functional notation for term constructors (partly because Agda's mixfix notation doesn't allow the usual terminal symbols of the λ-calculus) but it is otherwise a reasonably direct formalization of the original definition.

```
data Var : Set where
  x : ℕ → Var -- variables
_==_ : Var → Var → Bool
x n == x n′ = (n ≡ᵇ n′)
```

```
data Exp : Set where
  var_ : Var → Exp          -- variable value
  lam  : Var → Exp → Exp    -- lambda abstraction
  app  : Exp → Exp → Exp    -- application
```

Domains. The standard denotational semantics of the λ-calculus is based on a domain D_∞ isomorphic to the domain of all continuous functions from D_∞ to D_∞.[1] Its lightweight formalization postulates[2] the existence of an Agda type D∞ with a bijection _↔_ to the type D∞ → D∞ of *all* Agda functions on D∞.

```
open import Function
  using (Inverse; _↔_) public
open Inverse {{ ... }}
  using (to; from) public
```

```
postulate
  D∞ : Set
postulate
  instance iso : D∞ ↔ (D∞ → D∞)
```

[1] In fact the least solution of the domain equation $D_\infty = D_\infty \to D_\infty$ is a 1-element domain; the intended solution includes some arbitrary non-trivial domain.

[2] Assumptions are specified as postulates to avoid module parameters that would need to be repeated in importing modules, and to allow assumed properties to be added as rewrite rules.

The special module application Inverse {{ ... }} above has the effect of declaring the functions to : $D_\infty \to (D_\infty \to D_\infty)$ and from : $(D_\infty \to D_\infty) \to D_\infty$ to be inverse.

Domain equations in the denotational semantics of other languages generally involve also some flat domains, and domain constructors for cartesian product and separated sum. Their lightweight formalizations import standard Agda library modules for the corresponding datatypes and type constructors, and postulate groups of types with bijections to Agda type terms, as illustrated for PCF and Scheme in [11].

Environments are functions from the abstract syntax of variables to values in the domain D_∞. Ordering them pointwise defines a domain of environments. The lightweight formalization of this non-recursive domain in Agda is a simple type definition, together with the definition of the conventional notation for extending an environment with a single binding:

```
Env = Var → D∞

_[_/_] : Env → D∞ → Var → Env
ρ [ d / v ] = λ v′ → if v == v′ then d else ρ v′
```

Semantic functions. A conventional denotational semantics declares semantic functions from abstract syntax to domains of denotations, and defines the functions compositionally by semantic equations. Agda formalization of semantic functions is straightforward, as semantic equations can be written directly in Agda, and the type-checker reports any missing or overlapping cases. Some minor lexical adjustments to λ-notation are needed: $\lambda x.fx$ becomes $\lambda\ x \to f\ x$, adjacent names have to be separated by spaces, and sub- and superscript terms are not supported.

```
⟦_⟧ : Exp → Env → D∞
⟦ var v ⟧ ρ = ρ v

⟦ lam v e   ⟧ ρ = from ( λ d → ⟦ e ⟧ (ρ [ d / v ]) )
⟦ app e₁ e₂ ⟧ ρ = to ( ⟦ e₁ ⟧ ρ ) ( ⟦ e₂ ⟧ ρ )
```

Conventional denotational definitions usually elide the isomorphisms between domains and their definitions, but Agda requires explicit use of to and from in the formalization (cf. [17, §10.5]). The type-checker reports where elided isomorphisms need to be inserted.

Checking computed values. The following rewrite rule allows Agda to automatically evaluate the denotations of terms in the untyped λ-calculus, thereby supporting trivial proofs of equivalence. (Caveat: The proofs could be unsound, as the rewrite rule involves postulates.)

```
open Inverse using (inverseˡ)
to-from-elim : ∀ {f} → to (from f) ≡ f

to-from-elim = inverseˡ iso refl
{-# REWRITE to-from-elim #-}

check-convergence :
  ⟦ app (lam (x 1) (var x 42))
        (app (lam (x 0) (app (var x 0) (var x 0)))
             (lam (x 0) (app (var x 0) (var x 0)))) ⟧
  ≡ ⟦ var x 42 ⟧
check-convergence = refl

check-free :
  ⟦ app (lam (x 1)
          (app (lam (x 42) (var x 1))
               (var x 2)))
        (var x 42) ⟧ ≡ ⟦ var x 42 ⟧
check-free = refl
```

The denotational semantics of PCF involves explicit use of the fixed-point function fix. Its lightweight Agda formalization postulates fix f ≡ f (fix f). To use that property directly as a rewrite rule would lead to non-termination; however, the following derived property can be used, as it unfolds fix f only when f needs to be applied (as in *SIS* [8]): fix f p ≡ f (fix f) p.

Future work. It is clearly an *abuse* of Agda to type-check definitions based on potentially-unsound postulates. An implementation of some framework for (unguarded) SDT in Agda would presumably require a significant effort, but might contribute to increased interest in SDT, as well as providing proper foundations for lightweight formalization of denotational semantics.

Acknowledgments. Thanks to Benedikt Ahrens, Jesper Cockx, Tom de Jong, Bernhard Reus, Alex Simpson, and the 24th IFIP WG 2.11 meeting participants for helpful advice regarding formalization of denotational semantics and Agda, and to the anonymous reviewers for pertinent comments and suggestions for improvement.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 3: Semantic Structures*. Clarendon Press, 1994. URL https://achimjungbham.github.io/pub/papers/handy1.pdf.

[2] W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA 1989*, pages 433–443. ACM, 1989. doi:10.1145/74877.74922.

[3] M. P. Fiore and G. Rosolini. Domains in H. *Theor. Comput. Sci.*, 264(2):171–193, 2001. doi:10.1016/S0304-3975(00)00221-8.

[4] E. Giovannini, T. Ding, and M. S. New. Denotational semantics of gradual typing using synthetic guarded domain theory. *Proc. ACM Program. Lang.*, 9 (POPL), 2025. doi:10.1145/3704863.

[5] T. de Jong. TypeTopology/DomainTheory (Agda modules), since 2019. URL https://martinescardo.github.io/TypeTopology/DomainTheory.index.html.

[6] M. B. Kristensen, R. E. Møgelberg, and A. Vezzosi. Greatest HITs: Higher inductive types in coinductive definitions via induction under clocks. In *Proc. 37th Annual IEEE Symposium on Logic in Computer Science (LICS 2022)*, pages 42:1–42:13. IEEE Computer Society Press, 2022.

[7] J. R. Longley and A. K. Simpson. A uniform approach to domain theory in realizability models. *Math. Struct. Comput. Sci.*, 7(5):469–505, 1997. doi:10.1017/S0960129597002387.

[8] P. D. Mosses. SIS, 1979. URL https://pdmosses.github.io/software/sis/.

[9] P. D. Mosses. Agda code corresponding to a semantics of inheritance, 2024. URL https://github.com/pdmosses/jensfest-agda/.

[10] P. D. Mosses. Towards verification of a denotational semantics of inheritance. In *Proceedings of the Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday*, JENSFEST '24, page 5–13, New York, NY, USA, 2024. ACM. doi:10.1145/3694848.3694852.

[11] P. D. Mosses. Denotational semantics in Agda, 2025. URL https://github.com/pdmosses/xds-agda/. Experiments with lightweight formalization in Agda of existing language definitions.

[12] J. van Oosten and A. K. Simpson. Axioms and (counter) examples in synthetic domain theory. *Ann. Pure Appl. Log.*, 104(1-3):233–278, 2000. doi:10.1016/S0168-0072(00)00014-2.

[13] W. Phoa. Effective domains and intrinsic structure. In *LICS '90*, pages 366–377. IEEE Computer Society, 1990. doi:10.1109/LICS.1990.113762.

[14] B. Reus. Formalizing synthetic domain theory. *Journal of Automated Reasoning*, 23:411–444, 1999. doi:10.1023/A:1006258506401.

[15] B. Reus. Re: SDT in Agda. Personal communication, 2025.

[16] B. Reus and T. Streicher. General synthetic domain theory – a logical approach. *Math. Struct. Comput. Sci.*, 9(2):177–223, 1999. doi:10.1017/S096012959900273X.

[17] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[18] G. Rosolini. *Continuity and Effectivity in Topoi*. PhD thesis, Univ. of Oxford, 1986.

[19] Revised^5 report on the algorithmic language Scheme, 1998. URL https://standards.scheme.org/official/r5rs.pdf.

[20] D. Scott. Outline of a mathematical theory of computation. In *Proc. Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970. URL https://ncatlab.org/nlab/files/Scott-TheoryOfComputation.pdf. Also: Tech. Monograph PRG-2, Oxford Univ.

Computing Lab., Programming Research Group (1970). URL https://www.cs.ox.ac.uk/files/3222/PRG02.pdf.

[21] D. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136, Berlin, Heidelberg, 1972. Springer. doi:10.1007/BFb0073967. Also: Tech. Monograph PRG-7, Oxford Univ. Computing Lab., Programming Research Group (1971). URL https://www.cs.ox.ac.uk/files/3229/PRG07.pdf.

[22] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Inst. Symposia Series*, pages 19–46. Polytechnic Inst. of Brooklyn, 1971. Also: Tech. Monograph PRG-6, Oxford Univ. Computing Lab., Programming Research Group (1971). URL https://www.cs.ox.ac.uk/files/3228/PRG06.pdf.

[23] D. S. Scott. Relating theories of the lambda-calculus: Dedicated to Professor H. B. Curry on the occasion of his 80th birthday. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980. URL https://prl.khoury.northeastern.edu/blog/static/scott-80-relating-theories.pdf.

[24] A. Simpson. Computational adequacy for recursive types in models of intuitionistic set theory. *Annals of Pure and Applied Logic*, 130(1):207–275, 2004. doi:10.1016/j.apal.2003.12.005. Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS).

[25] A. K. Simpson. Computational adequacy in an elementary topos. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *CSL '98*, volume 1584 of *Lecture Notes in Computer Science*, pages 323–342. Springer, 1998. doi:10.1007/10703163_22.

[26] P. Stassen, R. E. Møgelberg, M. A. Zwart, A. Aguirre, and L. Birkedal. Modelling recursion and probabilistic choice in guarded type theory. *Proc. ACM Program. Lang.*, 9 (POPL), 2025. doi:10.1145/3704884.

[27] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, 1977.

[28] P. Taylor. The fixed point property in synthetic domain theory. In *LICS '91*, pages 152–160. IEEE Computer Society, 1991. doi:10.1109/LICS.1991.151640.

[29] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, Aug. 1976. doi:10.1145/360303.360308.

Author index

Abel, Andreas, 243
Adjedj, Arthur, 274
Affeldt, Reynald, 124
Ahrens, Benedikt, 59
Allais, Guillaume, 165
Almeida, Ana Jorge, 34
Altenkirch, Thorsten, 90
Alves, Sandra, 34
Awodey, Steve, 263

Bailitis, Janis, 29
Baillon, Martin, 9
bbchallenge Collaboration, 20
Benjamin, Thibaut, 274
van den Berg, Benno, 134
Berry, David G., 43
Bezem, Marc, 74
Blechschmidt, Ingo, 1
Bouverot-Dupuis, Mathis, 170
Bradley, Felix, 137
Bronsveld, Steven, 234
Bryant, Harry, 186
Buchholtz, Ulrik, 99
Bøgsted Poulsen, Danny, 17

Carneiro, Mario, 263, 271
Chapman, James, 198
Cherubini, Felix, 278
Chu, Fernando, 49
Coquand, Thierry, 278

Dagnino, Francesco, 194
Danielsson, Nils Anders, 243
Doré, Maximilian, 230
Díaz, Tomás, 82
Díaz-Caro, Alejandro, 147

Enriquez Mendoza, Javier, 183
Eriksson, Oskar, 243
Escardó, Martín H., 13, 96

Felicissimo, Thiago, 140
Ferrarini, Alessio, 260
Fiore, Marcelo, 43
Florido, Mário, 34
Forster, Yannick, 29, 170, 267
Francalanza, Adrian, 190
Fujiwara, Makoto, 204

Garrigue, Jacques, 124
Geerligs, Freek, 278
Geuvers, Herman, 215, 234
Ghani, Neil, 257
Giannini, Paola, 194
Gonçalves, April, 40
Gondelman, Léon, 17
Gratzer, Daniel, 99
Grotenhuis, Lide, 144

Hazratpour, Sina, 263
Herbelin, Hugo, 93
Hua, Joseph, 263
Huang, Yulong, 222
Hughes, Calum, 65
Hutton, Graham, 154
Hyvernat, Pierre, 257

Jack, Tom, 102
Jaskelioff, Mauro, 198
de Jong, Tom, 13, 281

Ka I Pun, Violet, 194
Kaposi, Ambrus, 90
Kapulkin, Krzysztof, 284
Kawakami, Ryuji, 124
Kirst, Dominik, 29, 116
Kocsis, Bálint, 127
Kokke, Wen, 40
Kovács, András, 207
Kraus, Nicolai, 281
Krebbers, Robbert, 127

Laretto, Andrea, 55
Lawrence, Andrew, 186
Lennon-Bertrand, Meven, 274
Leray, Yann, 131
Li, Yufeng, 284
Ljungström, Axel, 102
Loregian, Fosco, 55
Luo, Zhaohui, 137

Mahboubi, Assia, 9
Maillard, Kenji, 26, 274
Marin, Sonia, 2
Martens, Chris, 3
McBride, Conor, 37
Melkonian, Orestis, 198
Mihejevs, Zanzi, 201
Milner, Owen, 158
Moeneclaey, Hugo, 278
Monnier, Stefan, 86
Mosses, Peter D, 286

de Muijnck-Hughes, Jan, 174

Najmaei, Niyousha, 59
Nawrocki, Wojciech, 263
Neumann, Jacob, 52
North, Paige Randall, 49, 59

Otten, Daniël, 144

Pédrot, Pierre-Marie, 9
Pezlar, Ivo, 211
Poiret, Josselin, 23
Pradic, Cecilia, 253
Price, Ian, 253
Pujet, Loïc, 250
Punčochář, Vít, 211

Rahli, Vincent, 66, 183
Ramachandra, Ramkumar, 93
Restall, Greg, 110
Rice, Alex, 177
da Rocha Paiva, Bruno, 66
Rosain, Johann, 82, 218
Roux, Cody, 151
Rowicki, Radosław Jan, 190
Rydhof Hansen, René, 17

Sabelli, Pietro, 180
Saikawa, Takafumi, 124
Sattler, Christian, 4, 207
Scalas, Alceste, 190
Seisenberger, Monika, 186
Setzer, Anton, 186
Shahin, Ramy, 161
Shillito, Ian, 116
Sinkarovs, Artjoms, 257
Somayyajula, Siva, 120
Sozeau, Matthieu, 74, 82, 218
Speight, Sam, 183
Stepanenko, Sergei, 70
Stérin, Tristan, 20
Stump, Aaron David, 78
Ståhl, Casper, 17
Swan, Andrew Wakelin, 62

Tabareau, Nicolas, 82, 140
Takahashi, Yuta, 5
Tan, Yee-Jian, 267
Theocharis, Constantine, 240
Timany, Amin, 70
Torrella, Ulises, 194
Tosun, Ayberk, 96

Vazou, Niki, 260
Veltri, Niccolò, 55
Verkoelen, Joep, 215
Videla, André, 246

Watters, Sean, 46
van der Weide, Niels, 59, 106, 234
Weinberger, Jonathan, 99
Wiesnet, Franziskus, 225

Wilshaw, Sky, 154
Winterhalter, Théo, 82, 218
Wood, James, 237
Wullaert, Kobe, 106

Xu, Yiming, 26

Yallop, Jeremy, 222