25th International Conference on Types for Proofs and Programs

TYPES 2019

Oslo, Norway, 11 June - 14 June 2019

Abstracts

Centre for Advanced Study (CAS),
The Norwegian Academy of Science and Letters

Oslo, 2019

25th International Conference on Types for Proofs and Programs,
TYPES 2019
Oslo, Norway, 11 June - 14 June 2019
Abstracts
https://cas.oslo.no/types2019/

Edited by Marc Bezem, Niels van der Weide

Centre for Advanced Study (CAS),
The Norwegian Academy of Science and Letters
Drammensveien 78, 0271 Oslo, Norway
cas@cas.oslo.no

Preface

This volume contains the abstracts of the talks presented at the *25th International Conference on Types for Proofs and Programs, TYPES 2019* in Oslo, Norway, 11–14 June 2019.

The TYPES meetings are a forum to present new and on-going work in all aspects of type theory and its applications, especially in formalised and computer assisted reasoning and computer programming. The meetings from 1990 to 2008 were annual workshops of a sequence of five EU funded networking projects. Since 2009, TYPES has been run as an independent conference series. Previous TYPES meetings were held in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal near Nijmegen (2002), Torino (2003), Jouy-en-Josas near Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Torino (2008), Aussois (2009), Warsaw (2010), Bergen (2011), Toulouse (2013), Paris (2014), Tallinn (2015), Novi Sad (2016), Budapest (2017), and Braga (2018).

The TYPES areas of interest include, but are not limited to: foundations of type theory and constructive mathematics; applications of type theory; dependently typed programming; industrial uses of type theory technology; meta-theoretic studies of type systems; proof assistants and proof technology; automation in computer-assisted reasoning; links between type theory and functional programming; formalizing mathematics using type theory.

The TYPES conferences are of open and informal character. Selection of contributed talks is based on short abstracts; reporting work in progress and work presented or published elsewhere is welcome. A formal post-proceedings volume is prepared after the conference; papers submitted to that volume must represent unpublished work and are subjected to a full peer-review process.

TYPES 2019 was held in parallel with HoTT-UF, the workshop on Homotopy Type Theory and Univalent Foundations, 12–14 June 2019, in Oslo (cas.oslo.no/hott-uf). Wednesday 12 June the two events had a joint programme. Both events were part of the Special Year 2018/19 on Homotopy Type Theory and Univalent Foundations at the Centre for Advanced Study (CAS) at the Norwegian Academy of Science and Letters (cas.oslo.no/research-groups/homotopy-type-theory-and-univalent-foundations-article2083-827.html).

CAS provided generous support, both administrative and financial, which we gratefully acknowledge. We are also grateful for the support of COST Action CA15123 EUTypes. Finally, TYPES 2019 and HoTT-UF are also part of the project Computational Aspects of Univalence 2015-2020, led by Bezem and Dundas, and supported by the Research Council of Norway. We want to express our gratitude for the support by the RCN, which has been indispensable for establishing these research activities in Norway.

The combined events TYPES 2019 and HoTT-UF gathered 115 participants from almost 20 countries. The contributed part of the TYPES 2019 programme consisted of 50 talks. In addition, TYPES 2019 had four invited lectures, and three more from HoTT-UF on the day with the joint programme. (HoTT-UF had two days of additional invited lectures.)

Similarly to the 2011 and the 2013-2018 editions of the conference, the post-proceedings of TYPES 2019 will appear in Dagstuhl's *Leibniz International Proceedings in Informatics (LIPIcs)* series.

June 2019, Oslo Marc Bezem

Organisation

Program Committee

Thorsten Altenkirch	University of Nottingham
Marc Bezem	University of Bergen
Małgorzata Biernacka	University of Wrocław
Jesper Cockx	Chalmers University of Technology & University of Gothenburg
Herman Geuvers	Radboud University Nijmegen
Silvia Ghilezan	Univerity of Novi Sad
Mauro Jaskelioff	National University of Rosario
Ambrus Kaposi	Eötvös Loránd University
Ralph Matthes	IRIT, CNRS, University of Toulouse
Étienne Miquey	INRIA, University of Nantes
Leonardo da Moura	Microsoft Research
Keiko Nakata	SAP Potsdam
Fredrik Nordvall Forsberg	University of Strathclyde
Benjamin Pierce	University of Pennsylvania
Elaine Pimentel	Federal University of Rio Grande do Norte
Luís Pinto	University of Minho
Simona Ronchi Della Rocca	University of Turin
Carsten Schürmann	IT University of Copenhagen
Wouter Swierstra	Utrecht University
Tarmo Uustalu	Reykjavik University

Organising committee

Marc Bezem, Bjørn Ian Dundas, Erna Kas, and Camilla K. Elmar

Host

Centre for Advanced Study at the Norwegian Academy of Science and Letters

Sponsors

COST Action CA15123 EUTypes
Research Council of Norway
Academia Europaea Knowledge Hub Region Bergen

Table of Contents

An alphabetical index of all (co-)authors can be found on page 114.

Cubical Indexed Inductive Types

Evan Cavallo[1]

Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
ecavallo@cs.cmu.edu

Higher inductive types (HITs) provide a mutual generalization of inductive and quotient types and are an essential component of HoTT. However, the development of a general theory of HITs has lagged behind the use of specific instances; indeed, some examples employed in the HoTT book are still not known to have models in simplicial sets. I'll present my work with Robert Harper on giving a general schema for and computational interpretation of higher inductive types as part of a cubical type theory. Our schema includes indexed inductive types, which are non-trivial in cubical type theory even without higher constructors. On the way, I'll discuss why cubical type theory is especially well-suited for practical reasoning with HITs, and which problems remain open in cubical and other settings.

Challenges Scaling Type-Theory-Based Verification to Cryptographic Code in Production

Adam Chlipala[1]

MIT CSAIL
adamc@csail.mit.edu

There is a clear basic recipe for using dependent type theory in a proof assistant for proofs of programs, but the devil is in the details. I will discuss experiences overcoming practical challenges in applying Coq to generate efficient, verified cryptographic code. The result is code with about a billion users today, deployed in the Chrome web browser and elsewhere for small but important parts of negotiating secure (TLS) network connections. I will survey the context of cryptographic implementation and then turn closer to the hearts of most TYPES participants, reviewing basic questions of splitting functionality between a kernel type checker and tactics that generate proof terms. We found that the split embodied in Coq provided unacceptable performance when using partial evaluation to produce thousands of lines of C code in one go, which set us down the path of exploring an alternative approach to flexible term reduction and rewriting, while requiring even less sophistication than today's Coq kernel builds in. I'll explain the main insights behind the design of that system and how we apply it to produce what is essentially a verified domain-specific compiler for big-integer modular arithmetic.

Homotopy Canonicity for Cubical Type Theory

Simon Huber[1]

University of Gothenburg, Sweden
simonhu@chalmers.se

Cubical type theory provides a constructive justification of homotopy type theory and satisfies canonicity: every natural number is convertible to a numeral. A crucial ingredient of cubical type theory is a path lifting operation which is explained computationally by induction on the type involving several non-canonical choices. In this talk I will present why if we remove these equations for the path lifting operation from the system, we still retain homotopy canonicity: every natural number is path equal to a numeral. The proof involves proof relevant computability predicates (also known as sconing) and doesn't involve a reduction relation.

This is joint work with Thierry Coquand and Christian Sattler.

Classical analysis in dependent type theory

Assia Mahboubi[1]

Inria, LS2N
Assia.Mahboubi@inria.fr

This talk will present an on-going project which aims at setting up a new corpus of formalized real and complex analysis, developed using the Coq proof assistant. It will describe the foundations and the design principles adopted in this project, highlight its current salient features, and discuss its perspectives.

This is a joint work in progress with Raynald Affeldt, Cyril Cohen, Marie Kerjean, Damien Rouhling, Pierre-Yves Strub.

Check the Box!

Conor McBride[1]

University of Strathclyde, Glasgow, United Kingdom

A bidirectional presentation of Cubical Type Theory.

Homotopy Canonicity

Chistian Sattler[1]

University of Gothenburg, Sweden
sattler@chalmers.se

Because of the univalence and function extensionality axioms, homotopy type theory does not enjoy canonicity, a property expected from a constructive type theory. Voevodsky's 'homotopy canonicity' conjecture states that a homotopical version of this property still holds: every closed element of the natural number type is homotopic to a numeral. So far, this was known only for 1-truncated homotopy type theory, shown by Shulman using Artin glueing along a groupoid-valued global sections functor. In this talk, I will present a full proof of homotopy canonicity.

This is joint work with Krzysztof Kapulkin.

A Dependently-Typed Core Calculus for GHC

Stephanie Weirich[1]

University of Pennsylvania
sweirich@cis.upenn.edu

In this talk, I will give an overview of a new typed intermediate language for the Glasgow Haskell Compiler (GHC) and our experience with its development. This design is derived from efforts to add dependent types to the Haskell language. This challenge has forced my collaborators and I to consider how dependency interacts with GHC's many language features, including nontermination, parametric polymorphism, type families, GADTs and safe coercions. In response to these constraints, we have developed a core calculus that specifies how dependent types can work in this context. Furthermore, to make sure that this core calculus is a firm foundation for Haskell, we have used the Coq proof assistant to verify its metatheoretic properties, including type soundness.

Normalization by Evaluation for Call-by-Push-Value

Andreas Abel[*] and Christian Sattler

Department of Computer Science and Engineering, Gothenburg University

Normalization by evaluation (NbE) [Berger and Schwichtenberg, 1991] is the interpretation of an (open) term of type A as value in a suitable model $[\![A]\!]$, followed by *reification* of the value to a normal form of type A. Functions f in $[\![A \Rightarrow B]\!]$ are reified as λ-abstractions whose bodies are obtained by *reflecting* a fresh variable of type A as value a in $[\![A]\!]$ and reifying the application $f\,a$ at type B. A suitable model that supports fresh variable generation are *presheaves* over the category of typing contexts Γ and order-preserving embeddings $\Gamma \subseteq \Gamma'$, where a base type o is interpreted as the presheaf $\mathsf{Ne}\ o$ of neutral normal forms of type o, and function types by the presheaf exponential aka Kripke function space [Coquand, 1993, Altenkirch et al., 1995].

NbE for sum types requires a refinement of the model, since reflection of a variable of type $A + B$ as a value in $[\![A + B]\!]$ requires case distinction in the model. One such refinement are *sheaves* [Altenkirch et al., 2001]; another is the use of a monad $\mathcal{C}$ [Filinski, 2001, Barral, 2008] in the category of presheaves for the interpretation of sum types: $[\![A+B]\!] = \mathcal{C}([\![A]\!]+[\![B]\!])$. The smallest such *"cover" monad* $\mathcal{C}$ are binary trees where leaves are the monadic unit aka return, and the nodes case distinctions over neutrals $\mathsf{Ne}\,(A_1+A_2)$ of sum type. When leaves are normal forms, the whole tree represents a normal form, thus, $\mathsf{runNf} : \mathcal{C}(\mathsf{Nf}\,A) \to \mathsf{Nf}\,A$ is trivial. This *running of the monad* on normal forms represents the algorithmic part of the sheaf condition on $\mathsf{Nf}\,A$ and extends as $\mathsf{run} : \mathcal{C}[\![A]\!] \to [\![A]\!]$ to all semantic types.

The given interpretation of sum types $[\![A + B]\!] = \mathcal{C}([\![A]\!] + [\![B]\!])$ corresponds to the call-by-name (CBN) lambda calculus with lazy constructors. NbE can also be performed in call-by-value (CBV) style, then the monad is placed in the codomain of function types: $[\![A \Rightarrow B]\!] = [\![A]\!] \Rightarrow \mathcal{C}[\![B]\!]$ [Danvy, 1996]. A systematic semantic analysis of CBN and CBV lambda-calculi has been pioneered by Moggi [1991] through translation into his computational lambda calculus; Filinski [2001] studied NbE for the latter calculus using the continuation monad. Moggi's work was continued and refined by Levy [2006] who subsumed CBV and CBN under his monadic call-by-push-value (CBPV) calculus. In this work, we study NbE for CBPV.

CBPV was designed to study lambda-calculus with effects. It separates types into *value* types P and *computation* types N, which we, in analogy to polarized lambda-calculus [Zeilberger, 2009] refer to as *positive* and *negative* types. Variables stand for values, thus, have positive types. The monad that models the effects is placed at the transition from values to computations $\mathit{Comp}\,P$, and computations can be embedded into values by *thunking* ($\mathit{Thunk}\,N$).

$$\begin{array}{lclcll} \mathsf{Ty}^+ & \ni & P & ::= & o \mid P_1 + P_2 \mid \mathit{Thunk}\,N & \text{positive type / value type} \\ \mathsf{Ty}^- & \ni & N & ::= & P \Rightarrow N \mid \mathit{Comp}\,P & \text{negative type / computation type} \end{array}$$

We restrict to a fragment of *pure* CBPV with a single positive connective, sum types $P_1 + P_2$, and a single negative connective, call-by-value function types $P \Rightarrow N$. While we have no proper effects, the evaluation of open terms requires the effect of case distinction over neutrals, modeled by a cover monad $\mathcal{C}$. In the following, we give inductive definitions of the presheaves of normal (Nf) and neutral normal forms (Ne) of our fragment of CBPV and a concrete, strong cover monad Cov.

$$\mathsf{var}\ \frac{\mathsf{Var}\ o\,\Gamma}{\mathsf{Nf}\ o\,\Gamma} \qquad \mathsf{thunk}\ \frac{\mathsf{Nf}\ N\ \Gamma}{\mathsf{Nf}\ (\mathit{Thunk}\ N)\,\Gamma} \qquad \mathsf{inj}_i\ \frac{\mathsf{Nf}\ P_i\,\Gamma}{\mathsf{Nf}\ (P_1 + P_2)\,\Gamma} \qquad \mathsf{ret}\ \frac{\mathsf{Cov}\ (\mathsf{Nf}\ P)\,\Gamma}{\mathsf{Nf}\ (\mathit{Comp}\ P)\,\Gamma} \qquad \mathsf{abs}\ \frac{\mathsf{Nf}\ N\ (\Gamma.P)}{\mathsf{Nf}\ (P \Rightarrow N)\,\Gamma}$$

[*]Supported by VR grant 621-2014-4864 and EU Cost Action CA15123.

$$\mathsf{force}\ \frac{\mathsf{Var}\,(\mathit{Thunk}\ N)\,\Gamma}{\mathsf{Ne}\ N\ \Gamma} \qquad \mathsf{app}\ \frac{\mathsf{Ne}\,(P \Rightarrow N)\,\Gamma \qquad \mathsf{Nf}\ P\ \Gamma}{\mathsf{Ne}\ N\ \Gamma} \qquad \mathsf{bind}\ \frac{\mathsf{Ne}\,(\mathit{Comp}\ P)\,\Gamma \qquad \mathsf{Cov}\ \mathcal{J}\,(\Gamma.P)}{\mathsf{Cov}\ \mathcal{J}\ \Gamma}$$

$$\mathsf{return}\ \frac{\mathcal{J}\ \Gamma}{\mathsf{Cov}\ \mathcal{J}\ \Gamma} \qquad \mathsf{case}\ \frac{\mathsf{Var}\,(P_1 + P_2)\,\Gamma \qquad \mathsf{Cov}\ \mathcal{J}\,(\Gamma.P_1) \qquad \mathsf{Cov}\ \mathcal{J}\,(\Gamma.P_2)}{\mathsf{Cov}\ \mathcal{J}\ \Gamma}$$

($\mathcal{J}$ stands for an arbitrary presheaf in $\mathsf{Cov}\ \mathcal{J}$.) Normal forms start from a variable of base type and continue with introductions, except that the services of the monad can be used at the transition ret from positive to negative types ($\mathit{Comp}\ P$). Neutrals are eliminations of variables of type $\mathit{Thunk}\ N$ into a positive type $\mathit{Comp}\ P$, and can then be bound to a variable of type P to be used in a computation (see bind). Variables of sum type $P_1 + P_2$ can be utilized in computations through a case split.

Terms Tm of CBPV are obtained by blurring the distinction between Ne and Nf, generalizing bind and case from $\mathsf{Cov}\ \mathcal{J}$ to computations $\mathsf{Tm}\ N$, and relaxing var to variables of arbitrary type P and force to arbitrary terms of type $\mathit{Thunk}\ N$. Terms are evaluated in the following presheaf model, which interprets *Thunk* as the identity and *Comp* as Cov.

$$\begin{array}{lcl@{\qquad}lcl}
[\![P_1 + P_2]\!] & = & [\![P_1]\!] \mathbin{\hat{+}} [\![P_2]\!] & [\![P \Rightarrow N]\!] & = & [\![P]\!] \mathbin{\hat{\Rightarrow}} [\![N]\!] \\
[\![\mathit{Thunk}\ N]\!] & = & [\![N]\!] & [\![\mathit{Comp}\ P]\!] & = & \mathsf{Cov}[\![P]\!] \\
[\![o]\!] & = & \mathsf{Var}\ o & & &
\end{array}$$

The evaluation of bind terms in $\mathsf{Tm}\ N$ relies on $\mathsf{run} : \mathsf{Cov}[\![N]\!] \to [\![N]\!]$, which makes any computation type monadic. Reflection $\uparrow$ and reification $\downarrow$ are defined mutually by induction on the type. They take the usual form, only that reflection of positive variables is monadic, to allow the complete splitting of sums via case. It is invoked by reification of functions $\downarrow^{P \Rightarrow N}$ via runNf.

$$\begin{array}{lcl@{\qquad}lcl}
\uparrow^P & : & \mathsf{Var}\ P \to \mathsf{Cov}\ [\![P]\!] & \downarrow^P & : & [\![P]\!] \to \mathsf{Nf}\ P \\
\uparrow^N & : & \mathsf{Ne}\ N \to [\![N]\!] & \downarrow^N & : & [\![N]\!] \to \mathsf{Nf}\ N
\end{array}$$

The details of our construction, plus extension to product types and polarized lambda calculus, can be found in the full version at https://arxiv.org/abs/1902.06097. A partial Agda formalization is available at https://github.com/andreasabel/ipl.

References

T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In *CTCS'95*, vol. 953 of *LNCS*. Springer, 1995. https://doi.org/10.1007/3-540-60164-3_27.

T. Altenkirch, P. Dybjer, M. Hofmann, and P. J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS'01*. IEEE CS Press, 2001. https://doi.org/10.1109/LICS.2001.932506.

F. Barral. *Decidability for non-standard conversions in lambda-calculus.* PhD thesis, Ludwig-Maximilians-University Munich, 2008.

U. Berger and H. Schwichtenberg. An inverse to the evaluation functional for typed λ-calculus. In *LICS'91*. IEEE CS Press, 1991. https://doi.org/10.1109/LICS.1991.151645.

C. Coquand. From semantics to rules: A machine assisted analysis. In *CSL'93*, vol. 832 of *LNCS*. Springer, 1993. https://doi.org/10.1007/BFb0049326.

O. Danvy. Type-directed partial evaluation. In *POPL'96*. ACM, 1996. https://doi.org/10.1145/237721.237784.

A. Filinski. Normalization by evaluation for the computational lambda-calculus. In *TLCA'01*, vol. 2044 of *LNCS*. Springer, 2001. https://doi.org/10.1007/3-540-45413-6_15.

P. B. Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *HOSC*, 19(4), 2006. https://doi.org/10.1007/s10990-006-0480-6.

E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991. https://doi.org/10.1016/0890-5401(91)90052-4.

N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching.* PhD thesis, Carnegie Mellon University, 2009. http://software.imdea.org/~noam.zeilberger/thesis.pdf.

Bicategories in Univalent Foundations[*]

Benedikt Ahrens[1], Dan Frumin[2], Marco Maggesi[3], and Niels van der Weide[4]

[1] University of Birmingham, United Kingdom
b.ahrens@cs.bham.ac.uk
[2] Radboud University, Nijmegen, The Netherlands
dfrumin@cs.ru.nl
[3] University of Florence, Italy
marco.maggesi@unifi.it
[4] Radboud University, Nijmegen, The Netherlands
nweide@cs.ru.nl

In this work, we define and study bicategories in univalent foundations. More specifically, we define the notion of "univalent bicategory" and develop a mechanism to construct examples of such bicategories in a modular way.

Bicategories are category-like structures allowing for "morphisms between morphisms". They naturally arise when studying the model theory of type theory via "categories with structure" such as categories with families [4] and categories with attributes (see, *e.g.,* [7]). Other examples, such as groupoids and 1-types, are used in the study of the semantics of HITs [5].

By "univalent foundations", we mean the foundation given by univalent type theory (see, *e.g.,* the HoTT book [8]), with its notion of "univalent logic", and the anticipated interpretation of univalent type theory in simplicial sets arising from Voevodsky's simplicial set model [6].

In this model, univalent categories (just called "categories" in [2]), defined below, correspond to truncated complete Segal spaces, which in turn are equivalent to ordinary (set-theoretic) categories. This means that univalent categories are "the right" notion of categories in univalent foundations: they correspond exactly to the traditional set-theoretic notion of category. Similarly, the notion of *univalent bicategory*, studied in this work, provides the correct notion of bicategory in univalent foundations. Below, we explain what these notions mean precisely.

Univalent categories are categories for which the canonical maps

$$\mathsf{idtoiso}_{a,b} : a = b \to a \cong b,$$

sending equalities between objects a and b to isomorphisms, are equivalences. Univalent bicategories are defined analogously to univalent categories: we stipulate that the canonical maps

$$\mathsf{idtoiso}_0(a,b) : a = b \to a \simeq b \qquad \text{and} \qquad \mathsf{idtoiso}_1(f,g) : f = g \to f \cong g,$$

from identities on 0-cells a and b to adjoint equivalences, and from identities on 1-cells f and g to isomorphisms, respectively, are equivalences.

Showing that a bicategory is univalent can be difficult; this is particularly the case when the 0-cells of the bicategory are complicated structures obtained by layering data and properties—several such bicategories are mentioned later. In such a case, identities and adjoint equivalences between 0-cells are also complicated structures, and we would like to reason about the maps $\mathsf{idtoiso}_0$ and $\mathsf{idtoiso}_1$ modularly and "layerwise".

To this end, we develop the notion of *displayed bicategory* analogous to the 1-categorical notion of displayed category [3]. Intuitively, a displayed bicategory D over a bicategory B represents data and properties to be added to B to form a new bicategory: D gives rise to the *total bicategory* $\int \mathsf{D}$. Its cells are pairs (b, d) where d in D is a "displayed cell" over b in B.

[*]This work was supported by a grant from the COST Action EUTypes CA15123.

Let us consider an example. Take B to be the bicategory of 1-types, functions, and homotopies between them. Define a displayed bicategory D over B such that

1. displayed 0-cells over a 1-type are its points,
2. displayed 1-cells over $f : X \to Y$ and $x : X$ and $y : Y$ are paths $p_f : f(x) = y$, and
3. displayed 2-cells over a homotopy $\alpha : f \sim g$ are commutative triangles $\alpha_x \circ p_g = p_f$.

The total bicategory generated by this displayed bicategory is the bicategory of *pointed 1-types*.

Displayed bicategories can be iterated, thus yielding a convenient way to build complicated bicategories in layers. Can we reason layerwise to show that the resulting bicategory is univalent? Yes, we can, provided that each layer used to build the bicategory is "univalent" in a suitable sense. We introduce the notion of "(displayed) univalence" for displayed bicategories, a natural extension of the univalence condition for bicategories. Then we show

Result. *The total bicategory $\int \mathsf{D}$ of a displayed bicategory D over base B is **univalent** if B is univalent and D is univalent.*

Importantly, displayed "building blocks" can be provided, for which univalence is proved once and for all. These building blocks, *e.g.,* cartesian product, can be used like LEGO$^{\text{TM}}$ pieces to *modularly* build complicated bicategories that are automatically accompanied by a proof of univalence. We construct several such building blocks and show that they are univalent. We then use these blocks to construct a number of complicated univalent bicategories:

1. the bicategory of pseudofunctors between two univalent bicategories;
2. bicategories of algebraic structures; and
3. the bicategory of univalent categories with families.

Our definitions and results are formalized in the `UniMath` library of univalent mathematics. A full paper with more information is available [1].

References

[1] Benedikt Ahrens, Dan Frumin, Marco Maggesi, and Niels van der Weide. Bicategories in Univalent Foundations. arXiv:1903.01152, to appear in FSCD 2019.

[2] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science*, 25:1010–1039, 2015.

[3] Benedikt Ahrens and Peter LeFanu Lumsdaine. Displayed Categories *(conference version)*. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:16. Leibniz-Zentrum für Informatik, 2017.

[4] Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. *Mathematical Structures in Computer Science*, 24(6), 2014.

[5] Peter Dybjer and Hugo Moeneclaey. Finitary higher inductive types in the groupoid model. *Electr. Notes Theor. Comput. Sci.*, 336:119–134, 2018.

[6] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The Simplicial Model of Univalent Foundations (after Voevodsky), 2012.

[7] Andrew M. Pitts. Categorical Logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000.

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

Constructing Inductive-Inductive Types via Type Erasure

Thorsten Altenkirch[1], Ambrus Kaposi[2], András Kovács[2], and Jakob von Raumer[1]

[1] University of Nottingham, United Kingdom
thorsten.altenkirch@nott.ac.uk, jakob@von-raumer.de
[2] Eötvös Loránd University, Budapest, Hungary
{akaposi, kovacsandras}@inf.elte.hu

Inductive-inductive types [6, 4] allow the mutual definition of a type and, for example, a type family indexed over that type. This can be used to encode collections of types which are intricately inter-related such as the syntax of type theory itself, where we define a type $\mathsf{Con} : \mathcal{U}$ of contexts at the same time as a type of types over a context: $\mathsf{Ty} : \mathsf{Con} \to \mathcal{U}$. Thes types could be populated by an empty context $\mathsf{nil} : \mathsf{Con}$, a function for context extension $\mathsf{ext} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Con}$, a unit type former $\mathsf{unit} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}(\Gamma)$ and a former for Π-types of the form $\mathsf{pi} : (\Gamma : \mathsf{Con})(A : \mathsf{Ty}(\Gamma)) \to \mathsf{Ty}(\mathsf{ext}(\Gamma, A)) \to \mathsf{Ty}(\Gamma)$.

Many theorem provers like Coq [2] or Lean [3], which are based on foundations similar to the calculus of constructions (CoC), do not provide native support for inductive-inductive types. This raises the question about whether each example of an inductive-inductive type can be reduced to a (mutual) inductive family, which is supported by these kinds of system.

In the above example of contexts and types, we can achieve this goal by first stripping away the type dependencies ("type erasure") and then defining a predicate which states whether an instance of the erased types is well-formed according to the dependencies we erased (cf. last year's talk [1]). The recursor is obtained by defining a relation between erased types and the types of an arbitrary algebra `M`, which then can be shown to be functional. In Lean, the main definitions of such a construction would look like this, with `CT` being the erased types of contexts and types, `CTw` being their well-formedness predicate, and `rel` being the recursor relation:

```
inductive CT : bool → Type
| nil : CT ⊥
| ext : CT ⊥ → CT ⊤ → CT ⊥
| unit : CT ⊥ → CT ⊤
| pi : CT ⊥ → CT ⊤
    → CT ⊤ → CT ⊤
```

```
def CTw_arg (b : bool) : Type := if b then unit else CT ⊥

inductive CTw : Π b, CT b → CTw_arg b → Prop
| nil : CTw ⊥ CT.nil ()
| ext : Π Γ A xΓ, CTw ⊥ Γ xΓ → CTw ⊤ A Γ → CTw ⊥ (CT.ext Γ A) ()
| unit : Π Γ xΓ, CTw ⊥ Γ xΓ → CTw ⊤ (CT.unit Γ) Γ
| pi : Π Γ A B xΓ, CTw ⊥ Γ xΓ → CTw ⊤ A Γ → CTw ⊤ B (CT.ext Γ A)
    → CTw ⊤ (CT.pi Γ A B) Γ
```

```
def rel_arg (b : bool) : Type := if b then M.C else Σ γ, M.T γ

inductive rel : Π b, CT b → rel_arg b → Prop
| nil : rel ⊥ CT.nil M.nil
| ext : Π Γ A γ a, rel ⊥ Γ γ → rel ⊤ A ⟨γ, a⟩ → rel ⊥ (CT.ext Γ A) (M.ext γ a)
| unit : Π Γ γ, rel ⊥ Γ γ → rel ⊤ (CT.unit Γ) ⟨γ, M.unit γ⟩
| pi : Π Γ A B γ a b, rel ⊥ Γ γ → rel ⊤ A ⟨γ, a⟩ → rel ⊤ B ⟨M.ext γ a, b⟩
    → rel ⊤ (CT.pi Γ A B) ⟨γ, M.pi γ a b⟩
```

This raises the question about how to prove that this strategy works for *every possible inductive-inductive type* instead of specific examples. Compared to last year's contribution [1], considerable progress on this generalization has been made.

Dissecting the question, we provide answers for the following follow-up questions:

What are inductive-inductive types? We modify the approach of Kaposi and Kovács [5] to use the contexts of a domain-specific type theory to generate the signatures of inductive-inductive types. The syntax differentiates between *sort constructors* and *point constructors.*

What are inductive families? We use a similar approach to specify inductive families: We define a syntax of *sort contexts* and of *contexts over a sort context* to capture mutual inductive families which may be parameterized over metatheoretic types.

What assumptions do we postulate? We define notions of *displayed algebras and their sections* over contexts describing inductive families, such that we can formally denote the prerequesite of our reduction: for every such context, there exists an algebra (the constructor) which is initial in the sense that every displayed algebra over it has a section (the dependent eliminator).

How to define type erasure and well-formedness. We define two *syntactical translations*: One for the type erasure and, depending on an algebra over it, one for the inductively defined well-formedness predicate, like the one contained in the above code snippet.

How to define the initial algebra. Assuming algebras over these transformed contexts, we generate an algebra over the original inductive-inductive context of which we are confident that we will be able to show its initiality.

The above approach for constructing initial algebras differs from the term model construction in [5] in that we give a factorization of algebras into erased presyntax and well-formedness predicates. This can be seen as a first step towards a generic method of initiality proofs for various type theories presented as quotient inductive-inductive types. This would require to generalize the above steps to allow equality constructors in the syntax of signatures. We have formalized all results in Agda: `https://github.com/javra/indind-agda`

References

[1] Thorsten Altenkirch, Ambrus Kaposi, András Kovács, et al. Reducing inductive-inductive types to indexed inductive types. *TYPES 2018*, 2018.

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

[3] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015.

[4] Gabe Dijkstra. *Quotient inductive-inductive definitions.* PhD thesis, University of Nottingham, 2017.

[5] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages*, 3(POPL):2, 2019.

[6] Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University, 2013.

Containers of Applications and Applications of Containers

Malin Altenmüller and Conor McBride

University of Strathclyde, Glasgow, United Kingdom

We use the notion of indexed containers to define a datatype of *applications* in dependent type theory. Applications are server-like programs, indexed by their current status. The basis for the definition is the type of *interface* of an application, using the intuition that indexed containers are interaction structures [HH06]. The type of applications itself results from building cofree comonads on these containers. We take a comonadic view (applications provide a service) rather than a monadic view (applications do side effects), but Hancock, Setzer and Hyvernat's analysis of interaction structures [HS00, HSCS05] keep guiding our design. With indexed containers we additionally get both a notion of how applications display themselves to a user and combinators which enable modular construction of complex applications.

What is an Application? The applications we describe are *programs* which deliver some functionality to a user. The user sends requests to the application and the applications reacts accordingly. Applications act in a server-like way: they lazily await commands from their user, updating their internal state when receiving one. In addition they are always ready to display themselves to the user. Defining the type of applications consists of two parts: The type of specification (or interface) for applications contains the commands and responses an application can receive and send. The type of an application itself is based on an interface and coinductively defines its display type and its reaction to a command.

Indexed Containers as Templates. Hancock and Hyvernat [HH06] introduced indexed containers as *interaction structures* — protocols of communication between client and server. They consist of fields for commands and responses as well as for the next state the server will reach after a command-response pair has been sent. Indexed containers also represent indexed functors on the underlying indexed sets as defined by Altenkirch et al.[AGH+15]. Our type of specification for applications is a modified version of the description of interaction structure. Instead of computing the next status with an indexed function (depending on the response), we will use the contravariant powerset notion via a predicate on the type of status. This notion emphasises that only *some* properties of the server's next status might be known to the client, but not all of it. In Agda, the type of specification for applications looks like this:

```
record _◁_ (Now Next : Set) : Set₁ where
  field Commands : Now → Set
        Response  : (now : Now) → Commands now → (Next → Set)
```

Here *Now* and *Next* are the types of status an application can be in, the Commands an application can receive depend on its current status and the Response depends on the current status and the issued command and returns a property of the next status of the application. The closure properties of _◁_ are more flexible if *Now* and *Next* are separated, but we take fixpoints only when they coincide, i.e. when the container represents an *endofunctor*.

The Type of Application. With indexed containers as a notion of interface, we use them as the basis to define the type of applications themselves:

```
record App {S : Set} (spec : S ◁ S) (D : S → Set) (s : S) : Set where
  coinductive
  field display : D s
        react   : (c : spec .Commands s)
                  → (s : S) × (spec .Response s c s′ × App spec D s′)
```

We distinguish three types of *state* of applications: The static status S, the public display and a private internal state, which is the carrier of the coalgebra. When it receives a command, the application reacts by returning a Response and updating its internal state.

Having containers defining the interface of applications, we can use product operations on them to build higher dimensional structures and describe more complex applications. Examples include a text editor which is built from one-dimensional line editing applications and a window manager, controlling multiple overlapping windows on a screen.

The Type of Display. We also use indexed containers to define the type of *displays* of applications (called D in the above definition). This two-dimensional structure is indexed by its size and the underlying indexed container defines how to partition it. Using the size as index type makes the *pieces* (which result from partitioning) fit together properly. As we are describing structures which take up a limited amount of space, the underlying container is restricted to only have a *finite* amount of positions at which we can store data. Given a type of *tiles* (also indexed by their size) we can build interiors by plugging in the tiles into holes they fit in. This plugging-in operation is building the free monads of the underlying containers and gives us the type of display of an application. Again, combinators of the underlying containers let us build multi-dimensional structures. We start with notion of cutting a one-dimensional structure (i.e. dividing its length) and then use a product operation to get higher-dimensional structures. Partitioning a higher-dimensional structure consists of a choice of dimension, in which the split is performed, the other dimensions stay unchanged.

Summary. With these definitions of applications and displays we are building a library to construct complex applications from low dimensional ones. Using containers as the underlying structures results in a notion of combining containers by using product operations. We are currently implementing this library and aim for a sufficient collection of tools for building applications compositionally.

References

[AGH+15] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25:e5, 2015.

[HH06] Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137(1):189 – 239, 2006.

[HS00] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In Peter G. Clote and Helmut Schwichtenberg, editors, *Computer Science Logic*, pages 317–331, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[HSCS05] Peter Hancock, Anton Setzer, L Crosilla, and P Schuster. Interactive programs and weakly final coalgebras in dependent type theory. *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, 48:115–134, 2005.

Deep and Shallow Embeddings in Coq *

Danil Annenkov and Bas Spitters

Aarhus University

Abstract. We demonstrate how deep and shallow embeddings of functional programs can coexist in the Coq proof assistant using meta-programming facilities of MetaCoq. While deep embeddings are useful for proving meta-theoretical properties of a language, shallow embeddings allow for reasoning about the functional correctness of programs.

Motivation. Functional languages are becoming increasingly popular in software development. Moreover, there is a demand for languages with well-understood semantics arising from the concept of "smart contracts" — computer programs running on blockchains. A number of blockchain implementations have already adopted certain variations of functional languages as an underlying smart contract language. These languages range from minimalistic and low-level (Simplicity, Michelson) to fully-featured OCaml- and Haskell-like languages (Liquidity, Plutus). There are several formalisations of these languages in proof assistants, mostly covering meta-theory[1][2] with the notable exception of Scilla [2], which features verification of particular smart contracts translated by hand to Coq. One of the motivations for the present work is formalisation of the Concordium Oak functional smart contract language. Although it seems obvious to use the functional language of a proof assistant to represent constructions of a language we would like to reason about, there are not that many tools available for that purpose. We propose to use the meta-programming facilities of MetaCoq [1] to develop such a tool in a principled way.

Embedding of Functional Languages. There are various ways of reasoning about properties of a programming language in a proof assistant. First, let us split the properties in two groups: meta-theoretical properties (properties of a language itself) and properties of programs written in the language. Since we limit ourselves to a functional programming language, it is reasonable to assume that we can reuse the programming language of a proof assistant to express functional programs and reason about their properties. A somewhat similar approach is taken by the authors of the hs-to-coq library [3], which translates total Haskell programs to Coq by means of source-to-source transformation. Unfortunately, in this case it is impossible to reason about correctness of the translation.

An alternative is to directly interpret the syntax into Coq functions in NbE style [4]. In the presence of inductive types this requires encodings that complicate reasoning about the properties of the embedded programs. However, we do not rule out this approach, since it can be quite useful for proving meta-theoretical properties.

We would like: (1) to have a way of translating programs written in our functional language into a Coq function that looks close to the original (the shallow embedding); (2) to have access to the AST of the language for meta-theoretical reasoning (the deep embedding), and (3) to make an explicit connection between the semantics of the language and its representation in Coq in the form of a soundness theorem.

*This work is supported by the Concordium Blockchain Research Center, Aarhus University, Denmark.

[1]Michelson in Coq: `https://framagit.org/rafoo/michelson-coq/`

[2]Plutus meta-theory in Agda: `https://github.com/input-output-hk/plutus-metatheory`.

Our approach. MetaCoq makes it possible to connect the two ways of reasoning about functional programs. We implement a functional language that corresponds to a "core" subset of an average functional language with a System F type system, algebraic data types and general recursion. We define a translation from this language to the syntax of MetaCoq. In contrast with the source-to-source translation, our translation is a Coq function from the AST of our language to the AST of MetaCoq. This makes it possible to reason within Coq about the translation and formalize the required meta-theory for the language.

MetaCoq allows us to convert an AST represented as an inductive type into a Coq term. Thus, starting with the syntax of a program in our functional language, through a series of translations we produce a MetaCoq AST, which is then interpreted into a program in Coq's Gallina language. Let us consider the following example:

```
Definition plus_syn : expr := [| fix "plus" (x : Nat) : Nat → Nat :=
                                   case x : Nat return Nat → Nat of
                                   | Z → \y : Nat → y
                                   | Suc y → \z : Nat → Suc ("plus" y z) |].
(* Unquoting the syntax into a Coq term *)
Make Definition my_plus := Eval compute in (expr_to_term (indexify plus_syn)).
Lemma my_plus_correct n m : my_plus n m = n + m.
Proof. induction n;simpl;auto. Qed.
(* Computing with the interpreter *)
Compute (eval 10 enEmpty [| {plus_syn} 1 1 |]).
(* = Ok (vConstr "nat" "Suc" [vConstr "nat" "Suc" [vConstr "nat" "Z" []]]) *)
```

The term `plus_syn` defines an AST of a program in our functional language using the extension of the notation mechanism called *custom entries*. After the application of `indexify` (which converts variable names into De Bruijn indices) and `expr_to_term` (which translates expressions to the terms of MetaCoq) we use `Make Definition my_plus` := ... of MetaCoq to produce a Coq program. The `my_plus_correct` lemma shows that the `my_plus` corresponds to the standard definition of addition. This example already allows us to verify correctness of certain functions by proving them equivalent to functions from the standard library of Coq.

The semantics of our functional language is defined as a definitional interpreter in Coq. As the example above shows, this allows to compute with the interpreter on the deeply embedded representation. The interpreter is implemented in an environment-passing style and works both with named and nameless representations of variables. To be able to interpret general fixpoints we evaluate fixpoints applications in the environment extended with the closure corresponding to the recursive call. Due to the potential non-termination, we define our interpreter using a *fuel idiom*: by structural recursion on an additional argument (a natural number).

Since MetaCoq aims to also formalise the meta-theory of Coq we use this development to show that the semantics of our functional language agrees with its translation to MetaCoq (on terminating programs) and our interpreter is sound with respect to the embedding. This paves a way for principled embeddings of functional languages to Coq.

References

[1] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP18*, volume 10895 of *LNCS*, pages 20–39, 2018.

[2] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR*, abs/1801.00687, 2018.

[3] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. CPP18, pages 14–27. ACM, 2018.

[4] PawełWieczorek and Dariusz Biernacki. A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory. CPP 2018, pages 266–279. ACM, 2018.

Linear metatheory via linear algebra

Robert Atkey[1] and James Wood[1*]

University of Strathclyde, Glasgow, United Kingdom
{robert.atkey,james.wood.100}@strath.ac.uk

Introduction. We introduce a simply typed calculus $\lambda\mathcal{R}$ that allows the use of variables to be constrained by usage annotations in the context which binds them. $\lambda\mathcal{R}$ is a generalisation of existing core type theories for sensitivity analysis [RP10], dependency and confidentiality [ABHR99], linearity [Bar96], and modal validity [PD99]. It is related to quantitative type theory [Atk18], and various coeffect calculi [POM14, BGMZ14, GS14].

One of our insights is that because our usage annotations form a semiring, we have just enough structure to talk about vectors and matrices in the metatheory. We find useful some constructs of linear algebra, culminating in substitution phrased as application of a linear map.

An earlier version of this work was presented at TyDe 2018 [AW18]. The syntax and semantics are formalised in Agda, with the code at https://github.com/laMudri/quantitative/.

Syntax. Our syntax is that of a simply typed λ-calculus modified to let us reason about how variables are used. We assume a partially ordered semiring (posemiring) $(\mathscr{R}, \trianglelefteq, 0, +, 1, *)$ of usage annotations, with elements coloured in green for emphasis. The types are base types (ι_k), functions $(\multimap)$, tensor products $(1, \otimes)$, with products $(\top, \&)$, sums $(0, \oplus)$, and graded bangs $(!_\rho)$. A context $\Gamma^{\mathcal{R}}$ is the combination of a typing context Γ and a usage context $\mathcal{R}$. We see a usage context as a vector over $\mathscr{R}$ generated by the basis made of the variables it contains.

$$\rho, \pi \in \mathscr{R} \qquad A, B, C ::= \iota_k \mid A \multimap B \mid 1 \mid A \otimes B \mid \top \mid A \,\&\, B \mid 0 \mid A \oplus B \mid !_\rho A$$

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x : A \qquad \mathcal{P}, \mathcal{Q}, \mathcal{R} ::= \cdot \mid \mathcal{R}, x^\rho \qquad \Gamma^{\mathcal{R}} ::= \cdot \mid \Gamma^{\mathcal{R}}, x \overset{\rho}{:} A$$

Tensor products are eliminated by pattern matching (each side bound with annotation 1), whereas with products are eliminated by projections. The difference is correspondingly seen in the introduction rules, where the two halves of a tensor product have separate usage, and the two halves of a with product have shared usage (illustrated below).

The rule $\otimes$-I is the archetypal use of $+$. The constraint $\mathcal{P} + \mathcal{Q} \trianglelefteq \mathcal{R}$ says that $\mathcal{R}$ must be at least as permissive as the accumulation of usages in $\mathcal{P}$ and $\mathcal{Q}$. If the addition of the semiring is a join of the order (as in the modality example below), these two types of product are equivalent.

$$\frac{\Gamma^{\mathcal{P}} \vdash M : A \qquad \Gamma^{\mathcal{Q}} \vdash N : B \qquad \mathcal{P} + \mathcal{Q} \trianglelefteq \mathcal{R}}{\Gamma^{\mathcal{R}} \vdash (M, N)_\otimes : A \otimes B}\ \otimes\text{-I} \qquad \frac{\Gamma^{\mathcal{R}} \vdash M : A \qquad \Gamma^{\mathcal{R}} \vdash N : B}{\Gamma^{\mathcal{R}} \vdash (M, N)_\& : A \,\&\, B}\ \&\text{-I}$$

With the graded bang, we see use of $*$ from the annotation posemiring. We read $\rho * \pi$ as applying the action ρ to π. Introduction can be seen as scaling usage. Elimination is by pattern matching, where we bind a new variable with whatever usage annotation the type gave us.

$$\frac{\Gamma^{\mathcal{P}} \vdash M : A \qquad \rho * \mathcal{P} \trianglelefteq \mathcal{R}}{\Gamma^{\mathcal{R}} \vdash [M] : !_\rho A}\ !_\rho\text{-I} \qquad \frac{\Gamma^{\mathcal{P}} \vdash M : !_\rho A \qquad \Gamma^{\mathcal{Q}}, x \overset{\rho}{:} A \vdash N : B \qquad \mathcal{P} + \mathcal{Q} \trianglelefteq \mathcal{R}}{\Gamma^{\mathcal{R}} \vdash \text{let } [x] = M \text{ in } N : B}\ !_\rho\text{-E}$$

*James Wood is supported by an EPSRC Studentship.

The VAR rule (not pictured) at x can only be used in a usage context $\mathcal{R}$ when x has a usage annotation as permissive as 1, and all other variables have annotation as permissive as 0. In other words, x can be used plainly, and all other variables can be discarded. This can be succinctly stated as the constraint $x^1 \trianglelefteq \mathcal{R}$, where x^1 is the xth basis vector.

Substitution. We have two admissible rules leading up to the substitution lemma — subusaging (SUBUSE) and weakening (WEAK) — stated below. In the language of linear algebra, weakening is embedding into a space of higher dimension.

Let $|-|$ denote the length of a context. Usage contexts are taken to be row vectors. A *substitution* σ from $\Gamma^{\mathcal{P}}$ to $\Delta^{\mathcal{Q}}$ comprises a $|\mathcal{Q}| \times |\mathcal{P}|$ matrix $\mathbf{\Sigma}$ such that $\mathcal{Q}\mathbf{\Sigma} \trianglelefteq \mathcal{P}$, and for each $(x : A) \in \Delta$, a term M_x such that $\Gamma^{x^1\mathbf{\Sigma}} \vdash M_x : A$. Then, the simultaneous substitution lemma is proven via the linearity of vector-matrix multiplication.

$$\text{SUBUSE}\;\frac{\Gamma^{\mathcal{P}} \vdash M : A \qquad \mathcal{P} \trianglelefteq \mathcal{Q}}{\Gamma^{\mathcal{Q}} \vdash M : A} \qquad \text{WEAK}\;\frac{\Gamma^{\mathcal{P}} \vdash M : A}{\Gamma^{\mathcal{P}}, \Delta^{\mathbf{0}} \vdash M : A} \qquad \frac{\Delta^{\mathcal{Q}} \vdash N : A \quad \sigma : \Gamma^{\mathcal{P}} \Rightarrow \Delta^{\mathcal{Q}}}{\Gamma^{\mathcal{P}} \vdash N[\sigma] : A}\;\text{SUBST}$$

The identity substitution, where each variable x is substituted by the term x, is witnessed by the identity matrix. We expect composition to be witnessed by matrix multiplication.

Specialisations. To demonstrate the applicability of $\lambda\mathcal{R}$, and give examples of usage posemirings, we show that certain instances are translatable to DILL [Bar96] and the modal type theory of Pfenning and Davies [PD99]. Future work is to give an equational theory for $\lambda\mathcal{R}$, and show that these translations form an isomorphism.

DILL is a linear type theory where contexts are split between unrestricted and linear variables. To model linearity, we introduce the $\{0, 1, \omega\}$ posemiring. Annotation 0 denotes non-use, 1 linear use, and ω unrestricted use. Addition and multiplication are like the corresponding natural number operations, with ω acting as an infinite element and $1 + 1 = \omega$ in lieu of a 2 element. The order is generated by $0 \trianglelefteq \omega$ and $1 \trianglelefteq \omega$, with no relation between 0 and 1. This says that unrestricted variables can be both discarded and used. We translate a DILL derivation of $\Gamma; \Delta \vdash t : A$ into a $\lambda\mathcal{R}$ derivation of $\Gamma^{\omega}, \Delta^{\mathbf{1}} \vdash M_t : A$. We translate DILL's unannotated ! into $!_{\omega}$. In the translation, we make use of WEAK to ignore 0-use variables introduced by usage separation. When translating the other way, we require that $!_0$ and $!_1$ do not occur in the derivation we are translating. We translate a $\lambda\mathcal{R}$ derivation of $\Gamma^{\omega}, \Delta^{\mathbf{1}}, \Theta^{\mathbf{0}} \vdash M : A$ into a DILL derivation of $\Gamma; \Delta \vdash t_M : A$. This makes use of DILL's Environment Weakening lemma to correct cases where a $\lambda\mathcal{R}$ subderivation was too precise about usage.

Pfenning and Davies' modal type theory is already stated in the form of usage annotations. A variable is annotated either *true* or *valid*. Furthermore, conclusions are only ever of *true* things. This suggests that *true* is the 1 of the posemiring, and we introduce an *unused* annotation to be the 0. The PD variable rule says that both *true* and *valid* assumptions are *true*, so we have *true* $\trianglelefteq$ *valid*. Furthermore, all assumptions can be discarded, so *unused* is the bottom of the order. Addition is the join of this order. The modality $\Box$ is translated to $!_{valid}$, which tells us that *valid* $* \pi =$ *valid* for $\pi \neq$ *unused*. *unused* and *true* are the annihilator and unit of $*$, respectively. Having these definitions in place, the translations are similar to those for DILL.

Semantics. We also have a semantics that captures the intensional properties of programs via families of Kripke indexed relations that refine a simple set-theoretic semantics. This allows us to reconstruct the semantic properties of calculi in prior work for sensitivity analysis [RP10], and dependency and confidentiality [ABHR99], as well as a new calculus for monotonicity analysis.

References

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A Core Calculus of Dependency. In *POPL '99*, pages 147–160, 1999.

[Atk18] Robert Atkey. The syntax and semantics of quantitative type theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom*, 2018.

[AW18] Robert Atkey and James Wood. Context constrained computation. In *3rd Workshop on Type-Driven Development (TyDe '18), Extended Abstract*, 2018.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical report, The University of Edinburgh, 1996.

[BGMZ14] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A Core Quantitative Coeffect Calculus. In *ESOP 2014*, pages 351–370, 2014.

[GS14] Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *ESOP 2014*, pages 331–350, 2014.

[PD99] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. In *Mathematical Structures in Computer Science*, page 2001, 1999.

[POM14] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *ICFP 2014*, pages 123–135, 2014.

[RP10] J. Reed and B. C. Pierce. Distance makes the types grow stronger. In P. Hudak and S. Weirich, editors, *ICFP 2010*, pages 157–168, 2010.

Simply RaTT
A Fitch-style modal calculus for reactive programming

Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg

IT University of Copenhagen (bahr,cgra,mogel@itu.dk)

Reactive programs are programs, such as servers or much control software, that engage in an ongoing dialogue with their environment producing output from input, typically without terminating. Functional Reactive Programming (FRP) aims to provide high-level abstractions allowing control flow in reactive programming to be described in a simple and direct way. The basic such abstractions are signals and events. A signal is a time-dependent value, and in the special case of time being given by a discrete global clock, this just amounts to a stream.

Recently a number of authors [5, 4, 6] have suggested using modal types for functional reactive programming. By encoding time steps using a delay modality $\bigcirc$, one can ensure that all programs are causal. This can then be combined with Nakano's fixed point operator of type $(\bigcirc A \to A) \to A$ to recursively define programs while maintaining productivity. This fixed point operator has been the topic of much research also in dependent type theory lately, but the functional reactive applications have different operational requirements, in particular the avoidance of space leaks and time leaks forces restrictions on the calculus.

Our long term goal is to construct a dependent type theory for reactive programming (Reactive Type Theory, RaTT) which provides operational guarantees on programs while being expressive enough for program properties to be expressed and proved in the type theory. We build on work by Krishnaswami [6], who proved the absence of space leaks in a modal calculus for FRP. As most modal calculi, it uses let-expressions for eliminating modal types, and these are generally considered undesirable in dependent type theory.

Simply RaTT

This talk presents Simply Typed Reactive Type Theory (Simply RaTT), a calculus for reactive programming based on the Fitch-style approach to modal types [3, 2], in which introduction and elimination are given by abstracting and adding tokens to a context, thus avoiding let-expressions. For example, the token associated with the modal operator $\bigcirc$ is $\checkmark$, and the introduction and elimination rules for this are

$$\frac{\Gamma, \checkmark \vdash t : A}{\Gamma \vdash \mathsf{delay}\, t : \bigcirc A} \qquad\qquad \frac{\Gamma \vdash t : \bigcirc A \qquad \Gamma, \checkmark, \Gamma' \vdash}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}\, t : A}$$

The type $\bigcirc A$ classifies computations that can be run in the next time-step producing elements of type A. Time-steps are represented by adding a $\checkmark$ to a context, and thus adv corresponds to running a delayed computation in the next time step. The calculus also features a modal operator $\square$, which is used for representing stable computations, i.e., computations that can be safely executed any time in the future. It has the associated token $\sharp$.

Compared to Krishnaswami's calculus, which uses a more standard dual-context approach to modal types, Simply RaTT represents a shifted notion of time-dependence. Terms in Simply RaTT can depend on variables from the past (those before a $\checkmark$) as well as from the present. By contrast, terms in Krishnaswami's calculus can depend on the result of executing delayed computations in the future.

Fitch-style modal calculi have been used in dependently typed languages for guarded recursion [1]. In these, the tokens are named ticks, which are used when proving properties of recursive programs. We therefore hope to be able to reason about reactive programs using the Fitch-style approach in a future extension of Simply RaTT with dependent types.

Operational semantics

Simply RaTT is equipped with an operational semantics inspired by Krishnaswami [6]. Terms execute relative to a store used for storing delayed computations. The store can consist of zero, one, or two heaps, i.e., mappings of locations to terms. The three cases are written as $\sigma = \bot$, $\sigma = \sharp\eta_L$ and $\sigma = \sharp\eta_N \checkmark \eta_L$, respectively. A term executing in one of the two last stores can store delayed computations in η_L. In the last case, delayed computations can be retrieved from η_N and executed. For example, the operational semantics for delay and adv are

$$\frac{\sigma \neq \bot \qquad l \notin \mathsf{dom}\,(\mathsf{later}(\sigma))}{\langle \mathsf{delay}\, t; \sigma\rangle \Downarrow \langle l; \sigma, l \mapsto t\rangle} \qquad \frac{\langle t; \sharp\eta_N\rangle \Downarrow \langle l; \sharp\eta'_N\rangle \qquad \langle \eta'_N(l); \sharp\eta'_N \checkmark \eta_L\rangle \Downarrow \langle v; \sigma'\rangle}{\langle \mathsf{adv}\, t; \sharp\eta_N \checkmark \eta_L\rangle \Downarrow \langle v; \sigma'\rangle}$$

where $\mathsf{later}(\sigma)$ refers to the η_L component of the store σ.

The above operational semantics implements single step evaluation of reactive programs. From this one can define multistep evaluation of terms. For example, to evaluate a stream t, one would execute it in two empty heaps $\langle t; \sharp\emptyset\checkmark\emptyset\rangle \Downarrow \langle n :: l; \sharp\eta_L \checkmark \eta_N\rangle$ to a value consisting of a number n now, and a reference l to a delayed computation representing the tail. The tail can then be evaluated by executing $\langle \mathsf{adv}(l); \sharp\eta_N \checkmark \emptyset\rangle$. Note that the heap η_L in this step has been garbage collected.

Our main theorem states that any program of type $\Box(\mathsf{Str}(\mathsf{Nat}))$ can be executed for arbitrarily many steps following the above procedure, producing sequences of arbitrary length. Likewise, programs of type $\Box(\mathsf{Str}(\mathsf{Nat}) \to \mathsf{Str}(\mathsf{Nat}))$ represent stream processors that can process arbitrarily long input in a causal way to produce output of the same length. Input is represented by placing references of the form $l \mapsto n :: l'$ in the heap, where n is the current value of the input stream, and l' is a dangling reference to the tail which is only available in the next time step. By garbage collecting the η_L heap in each step, space leaks (in the form of indefinite storage of input data) is avoided. The safety of this algorithm for evaluating stream processors is proved using a Kripke logical relation.

References

[1] P. Bahr, H. B. Grathwohl, and R. E. Møgelberg. The clocks are ticking: No more delays! In *LICS*, 2017.

[2] Ranald Clouston. Fitch-style modal lambda calculi. In *FoSSaCS*, 2018.

[3] F. B Fitch. *Symbolic logic, an introduction.* Ronald Press Co., New York, NY, USA, 1952.

[4] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *PLPV*, 2012.

[5] Wolfgang Jeltsch. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In *PLPV*, 2013.

[6] N. R. Krishnaswami. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *ICFP*, 2013.

Free Algebraic Theories as Higher Inductive Types

Henning Basold[1]*, Niels van der Weide[2], and Niccolò Veltri[3]

[1] CNRS, ENS Lyon
`henning.basold@ens-lyon.fr`
[2] Radboud University Nijmegen
`nweide@cs.ru.nl`
[3] IT University of Copenhagen
`nive@itu.dk`

In recent years, there has been an increasing interest in higher inductive types. There are several reasons behind this, like synthetic homotopy theory [5], implementation of rewrite rules [1], quotients [2], and other colimits. Here we are interested in applications of higher inductive types in universal algebra and category theory, and the possibility of extending inductive and coinductive types beyond strictly positive types. One of the most basic constructions in universal algebra is that of a free algebra. This construction can be carried out by using higher inductive types, as we will now briefly show.

We present some basic notions as Agda code. In what follows, we will work in Martin-Löf type theory with two basic universes $\mathcal{U}_0 : \mathcal{U}_1$. We will also require function extensionality, which can, however, be dropped if we restrict ourselves to finitary signature. The following record type defines a *signature* (also polynomial or container) in Agda.

```
record Signature  : 𝒰₁ where
  sym : 𝒰₀
  ar   : sym → 𝒰₀
```

From a signature Σ : Signature, we can construct the type Term Σ X of terms (W-type) over Σ with leaves labelled in X. This type comes with the obvious iteration principle that we denote by Term-iter. An *algebraic theory* is given by a signature and a set of equations between terms over that signature. These equations may have free variables but may not use other properties than equality on the variables. Thus, we represent equations as parametric binary relations [4], as in the following record type.

```
record AlgTheory : 𝒰₁ where
  sig  : Signature
  eqs : ∀ {X : 𝒰₀} → Rel (Term sig X)
```

Note that the requirement that the variable type X is in a fixed universe $\mathcal{U}_0$ will also fix the universe in which the algebras for a theory live. This is, however, not a severe constraint, as the induction principle still allows for large elimination.

Given an algebraic theory T : AlgTheory over a signature Σ, we can define what an algebra and an induction scheme for an algebra of T is. Algebras are given in two steps: First, we define pre-algebras that do not have to fulfil the equations of the theory, but only are an algebra for the functor induced by the signature Σ. The fact that the carrier of a pre-algebra is a set (in the sense of homotopy type theory) is technical necessity, which we would like to lift in the future. Note that this pre-algebra immediately extends to all terms by freeness. An actual algebra for T is then a pre-algebra that fulfils also the equations required by the theory. This idea is formalised in the following two record types.

*This work is supported by the European Research Council (ERC) under the EU's Horizon 2020 programme (CoVeCe, grant agreement No. 678157)

```
record PreAlgebra : 𝒰₁ where
    carrier     : 𝒰₀
    carrier-set : is-set carrier
    algebra : (s : sym Σ) (α : ar Σ s → carrier) → carrier

  algebra* : Term Σ carrier → carrier
  algebra* = Term-iter (λ x → x) algebra

record Algebra : 𝒰₁ where
    pre-algebra : PreAlgebra
    resp-eq     : ∀ {t u : Term Σ carrier} →
                    eqs t u → algebra* t == algebra* u
```

One can then also define an induction scheme for algebras of an algebraic theory, cf. [3]. With the basic definitions in place, we construct the initial algebra of an algebraic theory as higher inductive type with the following (type) constructors, where the function node* of type Term Σ (FreeAlgebra T) → FreeAlgebra T is the extension of node to terms, cf. algebra* above.

```
FreeAlgebra : (T : AlgTheory) → 𝒰₀
node        : (s : sym Σ) (α : ar Σ s → FreeAlgebra T) → FreeAlgebra T
eq          : ∀ {t u} → eqs t u → node* t == node* u
quot        : is-set (FreeAlgebra T)
```

This HIT comes with an iteration principle for T-algebras and an induction principle. These can be used to show that FreeAlgebra T is indeed the initial T-algebra. The principles are strong enough to define the free algebra functor and show it is the left adjoint of the forgetful functor. Examples of this construction include the free monoid, which can be shown to be equivalent to lists. `https://perso.ens-lyon.fr/henning.basold/code/AlgTheoryHIT`. An alternative development in UniMath can be found in [6].

Why would we be interested in such a construction of free algebras? First of all, we wish to formalise parts of universal algebras, like quotient algebras and the isomorphism theorems. Higher inductive types seem to provide the appropriate mechanism for such an endeavour. Moreover, once we can construct free algebras, we could also consider inductive and coinductive types over algebraic theories as an extension of strictly positive types. In particular, semantics of finitely branching transition systems could be obtained in the final coalgebra for the finite powerset functor, which is the free join-semilattice and therefore representable in our framework. Finally, we aim to extend the construction to HITs that are not just sets, but groupoids. This would enable us, for example, to construct free symmetric monoidal categories.

In the talk, we will present the algebras and the induction scheme in more detail, show applications of the above construction, and discuss future directions.

References

[1] T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. In R. Bodík and R. Majumdar, editors, *Proc. of POPL 2016*, pages 18–29. ACM, 2016.

[2] H. Basold, H. Geuvers, and N. van der Weide. Higher Inductive Types in Programming. *J.UCS*, David Turner's Festschrift – Functional Programming: Past, Present, and Future, 2017.

[3] C. Hermida and B. Jacobs. Structural Induction and Coinduction in a Fibrational Setting. *Information and Computation*, 145:107–152, 1997.

[4] C. Hermida, U. S. Reddy, and E. P. Robinson. Logical Relations and Parametricity - A Reynolds Programme for Category Theory and Programming Languages. *ENTCS*, 303:149–180, 2014.

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

[6] N. van der Weide and H. Geuvers. The Construction of Set-Truncated Higher Inductive Types. Submitted.

Game Forms for Coalition Effectivity Functions

Colm Baston and Venanzio Capretta

School of Computer Science, University of Nottingham
{colm.baston,venanzio.capretta}@nottingham.ac.uk

Introduction Coalition logic, introduced by Pauly,[1] is a multi-agent modal logic for reasoning about what groups of agents can achieve if they act collectively, as a coalition. The semantics for coalition logic is based on *game forms*, which are essentially perfect-information strategic games where the players act simultaneously. From a game form, we can derive an *effectivity function* which defines those subsets of outcomes that a particular coalition can guarantee, regardless of how all other players act.

Pauly proves that there is a set of properties, *playability*, that precisely describe when an arbitrary effectivity function is the effectivity function for some strategic game. Our goal is to formalise this equivalence in the logics of the type-theoretic proof assistants Coq and Agda. Proving the playability of an effectivity function that is derived from a game form is straightforward, provided that we develop good libraries for decidable subsets of agents and states. The other direction is more complex, requiring the construction of a game form from a playable effectivity function, then proving that the derived effectivity function is equivalent to the original. In addition to adapting it for type-theoretic formalisation, we simplify Pauly's construction for the second direction, and we give a sketch of this below.

Game Forms A game form G is a tuple $\langle N, \{\mathcal{A}_i\}_{i \in N}, S, o\rangle$ where: N is a finite, non-empty set of agents (for n agents, we simply use the natural numbers $\{0, \ldots, n-1\}$); $\{\mathcal{A}_i\}_{i\in N}$ is a family of non-empty sets of actions for each agent i (a *strategy profile* $\sigma : \Pi_{i\in N}\mathcal{A}_i$ is a choice of actions for every agent); S is a set of possible outcome states; o is a function $(\Pi_{i\in N}\mathcal{A}_i) \to S$ that selects an outcome for every strategy profile.

A coalition C is a decidable subset of N. Let $\sigma_C : \Pi_{i\in C}\mathcal{A}_i$ be a strategy profile for C and $\sigma_{\overline{C}} : \Pi_{i\in\overline{C}}\mathcal{A}_i$ a strategy profile for the complement coalition $\overline{C} = N \setminus C$. We denote by $\sigma_C \oplus \sigma_{\overline{C}}$ a global strategy profile σ which joins the actions of both coalitions.

The effectivity function for game form G is a function $E_G : \mathcal{P}_{\mathsf{dec}}(N) \to \mathcal{P}(\mathcal{P}_{\mathsf{dec}}(S))$ which associates each coalition with a set of *goals*: each goal is a decidable set of states that the coalition can achieve by working together; that is, $X \in E_G(C)$ iff there is a strategy profile for C that guarantees an outcome in X, no matter the counter-strategy for $\overline{C}$. The effectivity function for a game form G is therefore defined by:

$$E_G(C) = \{X \in \mathcal{P}_{\mathsf{dec}}(S) \mid \exists\sigma_C, \forall\sigma_{\overline{C}}, o(\sigma_C \oplus \sigma_{\overline{C}}) \in X\}$$

In the semantics of coalition logic, it is very convenient to work abstractly with an effectivity function rather than directly with the game definition. Therefore we need a characterisation of those effectivity functions that come from games.

Playable Effectivity Functions An effectivity function $E : \mathcal{P}_{\mathsf{dec}}(N) \to \mathcal{P}(\mathcal{P}_{\mathsf{dec}}(S))$ is playable iff it satisfies the following properties: For any $C \subseteq N$, $\varnothing \notin E(C)$; For any $C \subseteq N$, $S \in E(C)$; E is N-*maximal*: for any $X \subseteq S$, $\overline{X} \notin E(\varnothing) \Rightarrow X \in E(N)$; E is *outcome-monotonic*: for any $C \subseteq N$ and any $X_1 \subseteq X_2 \subseteq S$, $X_1 \in E(C) \Rightarrow X_2 \in E(C)$; E is *superadditive*: for any

[1]Marc Pauly, "A modal logic for coalitional power in games", *J. of Logic and Computation*, 12, 02 2002.

disjoint pair $C_1, C_2 \subseteq N$, and any pair $X_1, X_2 \subseteq S$, $X_1 \in E(C_1) \wedge X_2 \in E(C_2) \Rightarrow X_1 \cap X_2 \in E(C_1 \cup C_2)$.

Two more properties follow from from the above: E *is regular*: for any $C \subseteq N$ and any $X \subseteq S$, $X \in E(C) \Rightarrow \overline{X} \notin E(\overline{C})$; E is *coalition-monotonic*: for any $C_1 \subseteq C_2 \subseteq N$, $E(C_1) \subseteq E(C_2)$.

Proving that for a game form G, E_G is playable is just a routine question of checking the properties. The inverse requires that for every playable E we construct a game form G such that $E = E_G$.

Game Form Construction Given a playable effectivity function $E : \mathcal{P}_{\mathsf{dec}}(N) \to \mathcal{P}(\mathcal{P}_{\mathsf{dec}}(S))$ for some non-empty sets N and S, we construct a game form G such that $E = E_G$. The set of agents and the set of states are N and S respectively, so we just need to define a family of sets of actions $\{\mathcal{A}_i\}_{i \in N}$, and an outcome function o.

An action for an agent $i \in N$ consists of a choice of a coalition C that i would like to be part of, a goal X that i would like the coalition to aim for, a selected outcome $x \in X$, and a natural number t which will be used in determining which agent gets to make the final decision:

$$\mathcal{A}_i = \{\langle C, X, x, t\rangle \mid C \subseteq N, i \in C, X \in E(C), x \in X, t \in \mathbb{N}\}$$

Let a strategy profile σ be given: we have a choice $\sigma_i = \langle C_i, X_i, x_i, t_i\rangle$ for every $i \in N$. A coalition $C \subseteq N$ is called *σ-cooperative* if, for every $i \in C$, $C_i = C$ and, for every $i, j \in C$, $X_i = X_j$. Let $X_C = X_i$ for any $i \in C$. Intuitively, a coalition C is σ-cooperative if all its members want to be in the coalition and they agree on the goal X_C they want to aim for.

Let $\langle C_1, \ldots, C_m\rangle$ be all the non-empty σ-cooperative coalitions, and let C_0 be the set of agents that are not in a σ-cooperative coalition. $\langle C_0, \ldots, C_m\rangle$ is a partition of N. Define $X_{C_0} = S$ and

$$O(\sigma) = \bigcap_{k=0}^{m} X_{C_k} = \bigcap_{k=1}^{m} X_{C_k}$$

The outcome of the game will be defined to be a state in $O(\sigma)$. The choice of the specific state will depend again on σ. We use the numbers t_i to determine an agent that will make the final decision: let $d = (\sum_{i \in N} t_i) \mod |N|$. The outcome will be the state chosen by this agent, x_d. However, this is not guaranteed to be an element of $O(\sigma)$: it is an element of X_d which is a superset of $O(\sigma)$. In case it isn't we revert to an arbitrary choice function $H : \Pi_{X \in E(N)} X$. This exists constructively because by definition of playable effectivity function every $X \in E(N)$ is non-empty. We can prove that $O(\sigma) \in E(N)$, so we can define:

$$o(\sigma) = \begin{cases} x_d & \text{if } x_d \in O(\sigma) \\ H(O(\sigma)) & \text{otherwise} \end{cases}$$

Theorem. $E = E_G$

Proof. From left to right, we assume $X \in E(C)$ for some coalition C, and must show that $X \in E_G(C)$. Expanding the definition: $\exists \sigma_C, \forall \sigma_{\overline{C}}, o(\sigma_C \oplus \sigma_{\overline{C}}) \in X$. Define σ_C by setting for every $i \in C$, $C_i = C$ and $X_i = X$; x_i and t_i may be chosen arbitrarily. By definition, C is a σ-cooperative coalition, so it will be one of the classes in the partition used to define $O(\sigma_C \oplus \sigma_{\overline{C}})$. As $X_C = X$, it follows that $O(\sigma_C \oplus \sigma_{\overline{C}}) \subseteq X$, and $o(\sigma_C \oplus \sigma_{\overline{C}}) \in X$, as desired.

We must omit the finer details in the right to left direction. The non-trivial case, where $C \neq N$, relies on the playability properties for the construction of a counter-strategy $\sigma_{\overline{C}}$ such that $O(\sigma) \in E(C)$. For each $x \in O(\sigma)$, we are able to tweak x_j and t_j for an agent $j \in \overline{C}$ such that $o(\sigma) = x$, showing that $x \in X$ and $O(\sigma) \subseteq X$, proving by playability that $X \in E(C)$. $\square$

Coherence via big categories with families of locally cartesian closed categories

Martin Bidlingmaier[*]

Dept. of Computer Science, Aarhus University, Denmark.
mbidlingmaier@cs.au.dk

Locally cartesian closed (lcc) categories are natural categorical models of extensional dependent type theory [See84]. However, there is a slight mismatch: syntactic substitution is functorial and commutes strictly with type formers, whereas pullback is generally only pseudo-functorial and preserves universal objects only up to isomorphism. In response to this problem, several notions of models with strict pullback operations have been introduced, e.g. categories with families (cwf) [Dyb96], and coherence techniques have been developed to "strictify" weak models such as lcc categories and obtain models with functorial substitution [CGH14][LW15]. Using these methods, a biequivalence of lcc categories and extensional type theories was established [CD14], but a higher categorical analogue is currently only conjectured [Kap15].

This talk introduces big cwf of lcc categories, a novel coherence construction for extensional type theory. Because we rely not on strictification but on particularly *incoherent* replacements of lcc categories, we conjecture that our technique generalizes well to the higher categorical case, giving rise to an interpretation of a weak dependent type theory without nontrivial definitional equalities but strict substitution in arbitrary lcc quasi-categories [Kap15].

Our point of departure is the observation that, when working in type theory, changing the ambient context is akin to changing the base terms of the underlying theory. For example, proving $v : \sigma \vdash t : \tau$ is equivalent to proving $\cdot \vdash t : \tau$ in a type theory that was freely extended by a term v of type σ. We take the idea that contexts represent different type theories literally and assign to each context a separate model, i.e. a separate lcc category. We thus work *among* lcc categories instead of within a single one. Context extension then corresponds to freely adjoining an interpretation of a term to an lcc category.

We have to be careful, however, because substitutions commute strictly with type formers, whereas the usual notion of lcc functor is only guaranteed to preserve lcc structure up to isomorphism. Substitutions are thus interpreted as *strict* lcc functors, which preserve a canonical choice of lcc structure on the nose. To account for possibly non-strict lcc functors we will encounter in the proof of theorem 1, we restrict ourselves to *variable* lcc categories Γ, for which every lcc functor $\Gamma \to \Delta$ is uniquely isomorphic to a strict one. So as to allow for nontrivial interpretations of the initial context, we consider lcc categories under some cobase $\mathcal{C}$.

Definition 1. Let $\mathcal{C}$ be an lcc category. The *cwf of lcc categories under* $\mathcal{C}$ is given as follows.

- A context is a variable lcc functor $B_\Gamma : \mathcal{C} \to \Gamma$.
- A context morphism from $B_\Gamma : \mathcal{C} \to \Gamma$ to $B_\Delta : \mathcal{C} \to \Delta$ is a strict lcc functor $F : \Delta \to \Gamma$ such $FB_\Delta = B_\Gamma$.
- A type in context $B_\Gamma : \mathcal{C} \to \Gamma$ is an object of Γ.
- A term of type σ in context $B_\Gamma : \mathcal{C} \to \Gamma$ is a morphism $\top_\Gamma \to \sigma$ in Γ.
- Substitution along context morphisms is given by application of strict lcc functors.

[*]Supported by AFOSR grant 12595060.

Theorem 1. *The cwf of lcc categories under $\mathcal{C}$ has an initial context and comprehensions, and it supports Σ, Π and extensional identity types.*

Theorem 1 enables the interpretation of a type theory with the corresponding type formers in our cwf. The initial context is constructed by discarding the canonical choice of lcc structure of $\mathcal{C}$ and adjoining new canonical lcc structure. This structure satisfies only the equations that follow from the lcc axioms, so this construction can be understood as making $\mathcal{C}$ maximally incoherent. The initial context is equivalent to $\mathcal{C}$, so that all constructions in the empty context can be transported back into $\mathcal{C}$.

The existence of comprehensions follows from the following lemma, whose 2-categorical content (compare [BKP89]) is also the main ingredient in our solution to the coherence problem.

Lemma 1. *Let Γ be an lcc category and let $\sigma \in \operatorname{Ob}\Gamma$ be an object. Then there is a strict lcc functor $P_\sigma : \Gamma \to \Gamma.\sigma$ and a morphism $v : \top \to P_\sigma(\sigma)$ in $\Gamma.\sigma$ such that for every strict lcc functor $Q : \Gamma \to \Delta$ and morphism $w : \top \to Q(\sigma)$, there is a unique strict lcc functor $R = \langle Q, w\rangle : \Gamma.\sigma \to \Delta$ such that $R(v) = w$. Moreover, if $R_1, R_2 : \Gamma.\sigma \to \Delta$ are (not necessarily strict) lcc functors and $\phi : R_1 P_\sigma \overset{\sim}{\Rightarrow} R_2 P_\sigma : \Gamma \to \Delta$ is a natural isomorphism which is suitably compatible with v, then there is a unique natural isomorphism $\psi : R_1 \overset{\sim}{\Rightarrow} R_2$ such that $\psi P_\sigma = \phi$.*

To demonstrate the use of lemma 1, we sketch the construction of dependent function types $\Pi(\sigma, \tau)$ and function application $\mathrm{App}^\tau_\sigma(s, t)$. The pullback functor $\sigma^* : \Gamma \to \Gamma_{/\sigma}$ is lcc and thus corresponds to a strict lcc functor $(\sigma^*)^s$ by variability. Together with the diagonal $\sigma \to \sigma \times \sigma$, it induces a strict lcc functor $D_\sigma : \Gamma.\sigma \to \Gamma_{/\sigma}$ by the universal property of $\Gamma.\sigma$. We then define $\Pi(\sigma, \tau) = \Pi_\sigma(D_\sigma(\tau))$, where $\Pi_\sigma : \Gamma_{/\sigma} \to \Gamma$ denotes the right adjoint to σ^*.

Now if $s : \top \to \Pi(\sigma, \tau)$ and $t : \top \to \sigma$, it is straightforward to produce a term of type $t^*(D_\sigma(\tau))$, but we need a term of type $\bar{t}(\tau)$, where $\bar{t} = \langle \mathrm{Id}_\Gamma, t\rangle : \Gamma.\sigma \to \Gamma$. There is a canonical isomorphism $t^* \circ D_\sigma \overset{\sim}{\Rightarrow} \bar{t}$ which is constructed using lemma 1 from the ismorphism $t^* \circ \sigma^* \overset{\sim}{\Rightarrow} \mathrm{Id}_\Gamma : \Gamma \to \Gamma$. We may thus define

$$\mathrm{App}^\tau_\sigma(s, t) : \top \longrightarrow t^*(D_\sigma(\tau)) \xrightarrow{\sim} \bar{t}(\tau),$$

which has the appropriate type.

References

[BKP89] R. Blackwell, G.M. Kelly, and A.J. Power. Two-dimensional monad theory. *Journal of Pure and Applied Algebra*, 59:1–41, 1989.

[CD14] P. Clairambault and P. Dybjer. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. *Mathematical Structures in Computer Science*, 24(6):e240606, 2014.

[CGH14] P.-L. Curien, R. Garner, and M. Hofmann. Revisiting the categorical interpretation of dependent type theory. *Theoretical Computer Science*, 546:99–119, 2014. Models of Interaction: Essays in Honour of Glynn Winskel.

[Dyb96] P. Dybjer. Internal type theory. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '95, pages 120–134. Springer-Verlag, 1996.

[Kap15] Chris Kapulkin. Locally cartesian closed quasicategories from type theory. *Journal of Topology*, 10, 07 2015.

[LW15] P. L. Lumsdaine and M. A. Warren. The local universes model: An overlooked coherence construction for dependent type theories. *ACM Trans. Comput. Logic*, 16(3):23:1–23:31, July 2015.

[See84] R. A. G. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95(1):33–48, 1984.

Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

Frédéric Blanqui[1][2], Guillaume Genestier[2][3], and Olivier Hermant[3]

[1] INRIA
[2] LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay
[3] MINES ParisTech, PSL University

We introduce a termination criterion for a large class of programs whose operational semantics can be described by higher-order rewriting rules typable in the $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$ for short).

$\lambda\Pi/\mathcal{R}$ is a system of dependent types where types are identified modulo the β-reduction of λ-calculus and rewriting rules given by the user to define not only functions but also types. Those rewriting rules can be non-orthogonal, meaning that they can overlap or be non-linear. An example including overlapping and a type defined by rewriting rules is the propositional integer comparison:

```
symbol Prop : TYPE                     symbol ⊥ : Prop          symbol ⊤ : Prop
symbol Prf : Prop ⇒ TYPE               symbol infix ≤ : Nat ⇒ Nat ⇒ Prop
 rule Prf ⊤ ⟶ Πc:Prop. Prf c ⇒ Prf c      rule (s x) ≤ 0      ⟶ ⊥
 rule Prf ⊥ ⟶ Πc:Prop. Prf c              rule     0 ≤ y      ⟶ ⊤
 rule x ≤ x ⟶ ⊤                           rule (s x) ≤ (s y) ⟶ x ≤ y
```

Dependency pairs are a key concept at the core of modern automated termination provers for first-order term rewriting systems. Arts and Giesl [2] proved that a first-order rewriting relation terminates if and only if there are no infinite chains, that are sequences of dependency pairs interleaved with reductions in the arguments. We extend this notion of dependency pair to higher-order rewriting. Then we prove that, for a large class of rewriting systems $\mathcal{R}$, the combination of β and $\mathcal{R}$ is strongly normalizing on terms typable in $\lambda\Pi/\mathcal{R}$ if, there is no infinite chain.

To do so, we first construct a model of this calculus based on an adaptation of Girard's reducibility candidates [5], and prove that every typable term is strongly normalizing if every symbol of the signature is in the interpretation of its type (Adequacy lemma). We then prove that this hypothesis is verified if there is no infinite chain.

Our criterion has been implemented in SizeChangeTool. For now, it takes as input Xtc (used for the termination competition) or Dedukti files, but it could be easily adapted to a subset of other languages like Agda. As far as we know, this tool is the first one to automatically check termination in $\lambda\Pi/\mathcal{R}$, which includes both higher-order rewriting and dependent types.

Definition 1 ($\lambda\Pi/\mathcal{R}$)**.** $\lambda\Pi/\mathcal{R}$ is the PTS λP [3], enriched by a finite signature $\mathbb{F}$ and a set $\mathcal{R}$ of rules $(\Delta, f\,\vec{l} \to r)$ such that $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$ and Δ is a context associating a type to every variable of $\vec{l}$. Each $f \in \mathbb{F}$ has a type Θ_f and a sort s_f. f is (partially) defined if it is the head of the left-hand side of a rule.

Let $\to\; = \;\to_\beta \cup \to_\mathcal{R}$ where $\to_\beta$ is the β-reduction of λ-calculus and $\to_\mathcal{R}$ is the smallest relation containing $\mathcal{R}$ and closed by substitution and context. The typing rules are the ones of λP too, but the conversion is enriched with $\mathcal{R}$ and function symbol introduction is similar to variable introduction.

$$\text{(conv)}\ \frac{\Gamma \vdash a : A \qquad A \to^* {}^*\!\leftarrow B \qquad \Gamma \vdash B : s}{\Gamma \vdash a : B} \qquad \text{(fun)}\ \frac{\Gamma \vdash \Theta_f : s_f}{\Gamma \vdash f : \Theta_f}$$

We assume that $\to$ is locally confluent and preserves typing. For all f, we require $\vdash \Theta_f : s_f$.

As a β step can generate an $\mathcal{R}$ step, and vice versa, we cannot expect to prove the termination of $\to_\beta \cup \to_\mathcal{R}$ from the termination of $\to_\beta$ and $\to_\mathcal{R}$. The termination of $\lambda\Pi/\mathcal{R}$ cannot be reduced to the termination of the simply-typed λ-calculus either because of type-level rewriting rules.

So we build a model of our calculus by interpreting types into sets of terminating terms, adapting Girard's candidates [5]. To do so, we assume given a well-founded order on symbols of sort $\square$. This interpretation is such that if f is a function symbol of type $\Pi(\vec{x} : \vec{T}).U\,\vec{y}$, then $f\,\vec{x} \in [\![U\,\vec{y}]\!]$ implies $x_i \in [\![T_i]\!]$ if i is an accessible position in f (accessibility is similar to the constraint of positivity of inductive types, for rewriting; see [4] for a definition).

We then extend this definition to interpret all types, such that if $T \to U$, then $[\![T]\!] = [\![U]\!]$.

We use $\sigma \vDash \Gamma$ to denote that for all $x : T$ in Γ, $\sigma(x) \in [\![T]\!]_\sigma$.

Lemma 2 (Adequacy)**.** Assuming for all f, $f \in [\![\Theta_f]\!]$, if $\Gamma \vdash t : T$ and $\sigma \models \Gamma$, then $t\sigma \in [\![T]\!]_\sigma$.

The hypothesis "for all f, $f \in [\![\Theta_f]\!]$" can be reduced to the absence of infinite chains, as shown by Arts and Giesl for first-order rewriting [2].

Definition 3 (Dependency pairs)**.** Let $f\,\vec{l} > g\,\vec{m}$ iff there is a rule $f\,\vec{l} \to r \in \mathcal{R}$, g is (partially) defined and $g\,\vec{m}$ is a maximally applied subterm of r.

$f\,t_1 \ldots t_p \tilde{>} g\,u_1 \ldots u_q$ iff there are a dependency pair $f\,l_1 \ldots l_i > g\,m_1 \ldots m_j$ with $i \leq p$ and $j \leq q$ and a substitution σ such that, for all $k \leq i$, $t_k \to^* l_k\sigma$ and, for all $k \leq j$, $m_k\sigma = u_k$.

We consider a pre-order $\succeq$ on $\mathbb{F}$ compatible with typing and rewriting (*ie.* if $g \in \Theta_f$ or $r \rhd g$ for $f\,\vec{l} \to r \in \mathcal{R}$, then $f \succeq g$). $\mathcal{R}$ is well-structured and accessible if for every rule $(\Delta, f\,\vec{l} \to r)$, r is typable using only symbols smaller or equal to f and for every substitution σ, if $\Theta_f = \Pi\vec{x} : \vec{T}.U$, then $[\vec{x} \mapsto \vec{l}]\sigma \models \vec{x} : \vec{T}$ implies $\sigma \models \Delta$.

Theorem 4. The relation $\to = \to_\beta \cup \to_\mathcal{R}$ terminates on terms typable in $\lambda\Pi/\mathcal{R}$ if $\to$ is locally confluent and preserves typing, $\mathcal{R}$ is well-structured and accessible, and $\tilde{>}$ terminates.

Following [4], accessibility can be checked, by imposing that every variable occurring in the right-hand side of a rule is accessible in the left-hand side. Following [7], to prove that $\tilde{>}$ terminates, we can use Lee, Jones and Ben-Amram's size-change termination criterion [6].

Thus, we obtain a modular criterion extending Arts and Giesl's theorem that a rewriting relation terminates if there are no infinite chains [2] from first-order rewriting to dependently-typed higher-order rewriting.

This result also extends Wahlstedt's work [7] from weak to strong normalisation. Like Wahlstedt's work, AGDA's termination checker [1] is designed for rules defined by constructors pattern matching, enforcing the rewriting system to be orthogonal and every definition to be total. Our criterion requires a much weaker condition: local confluence.

References

[1] A. Abel. foetus – Termination Checker for Simple Functional Programs. 1998

[2] T. Arts, J. Giesl. Termination of term rewriting using dependency pairs. *TCS* 236:133–178, 2000.

[3] H. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science. Volume 2. Background: computational structures*, p. 117–309. Oxford University Press, 1992.

[4] F. Blanqui. Definitions by rewriting in the calculus of constructions. *MSCS* 15(1):37–92, 2005.

[5] J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and types*. Cambridge University Press, 1988.

[6] C. S. Lee, N. Jones, A. Ben-Amram. The size-change principle for program termination. POPL'01.

[7] D. Wahlstedt. *Dependent type theory with first-order parameterized data types and well-founded recursion*. PhD thesis, Chalmers University of Technology, 2007.

Weak Type Theory is Rather Strong

Simon Boulier and Théo Winterhalter

Gallinette Project-Team, Inria Nantes France

Abstract

We often put computation at the core of our theories, showing Strong Normalisation and confluence of our calculi. Thanks to the Curry-Howard isomorphism we know that computation corresponds to cut-elimination, or simplification of proofs. In practice it is also useful because it allows to identify expressions such as $0+n$ and n. We claim however that computation does not bring more logical power to our type theories by means of a Coq formalisation of a translation from Extensional Type Theory to a Weak Intensional Type Theory with axioms K and functional extensionality. Our result extends to the case of 2-level type theories, making it relevant in the homotopy setting.

Computation. The basic computation rule is the β-reduction (where square brackets represent substitution):

$$(\lambda x.\ t)\ u \equiv t[x := u]$$

In dependent type theories, computation is not limited to β-reduction and includes computation rules for eliminators of inductive types (or pattern-matching). We can also extend the notion and think of an equational theory that includes reduction but also η-rules or even more recently in the case of Coq and Agda, definitional proof-irrelevance of propositions [3].

$$\begin{array}{rlrl} f \equiv & \lambda x.\ f\ x & p \equiv & (\pi_1\ p, \pi_2\ p) \\ \mathsf{S}\ n + m \equiv & \mathsf{S}\ (n+m) & 0+n \equiv & n \\ (x : P : \mathsf{Prop}) \equiv & (y : P : \mathsf{Prop}) & x \equiv & \star : \mathsf{Unit} \end{array}$$

Weak type theories (WTT) are type theories without computation rules. Instead all of these rules are *weak*, i.e., assumed as equality axioms (for the type of propositional equality). For instance, β-reduction is represented by the following equality (where $x =_T y$ is the identity type representing propositional equality of x and y at type T).

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \beta(x.t, u) : (\lambda x.\ t)\ u =_{B[x:=u]} t[x := u]}$$

And the same goes for other equation rules. In particular we also make explicit the δ-reduction stating that a definition $id := t$ can be unfolded. We additionally require axioms K, functional extensionality and more surprising congruence rules for the other binders:

$$\frac{\Gamma \vdash p : t =_A t}{\Gamma \vdash \mathsf{K}(p) : p = \mathsf{refl}_A\ t} \qquad \frac{\Gamma, x : A \vdash p : f\ x = g\ x}{\Gamma \vdash \mathsf{funext}(x.p) : f = g}$$

$$\frac{\Gamma, x : A \vdash p : B_1 = B_2}{\Gamma \vdash \mathsf{cong}\Pi(x.p) : \Pi(x:A).B_1 = \Pi(x:A).B_2} \qquad \frac{\Gamma, x : A \vdash p : B_1 = B_2}{\Gamma \vdash \mathsf{cong}\Sigma(x.p) : \Sigma(x:A).B_1 = \Sigma(x:A).B_2}$$

Since our goal is to prove that extensional type theory (ETT) is conservative over WTT—every valid WTT type that is inhabited in ETT is inhabited in WTT—and since ETT proves these axioms, we have to assume them in WTT. The axioms $\mathsf{cong}\Pi$ and $\mathsf{cong}\Sigma$ do not seem to be derivable from funext.

Extensional Type Theory is distinguished from the usual Intensional Type Theory (ITT) by the reflection rule

$$\frac{\Gamma \vdash e : u =_A v}{\Gamma \vdash u \equiv v}$$

turning any provable equality into a conversion (i.e., computation rule in some sense). ITT trivially embeds into ETT so our result also extends to ITT to WTT.

Translating ETT to WTT. This work builds on previous work [5] of translating ETT to ITT: the idea is to take a typing derivation in ETT and produce a translation of the term (which was a decoration of the original term with rewriting of equalities) and a proof that it is typable in ITT.

This whole work has been formalised in Coq [2]. We proceed in two phases. For the first one, we take advantage of our previous formalisation by remarking that the way we deal with conversion in the target is mainly orthogonal to the translation itself. We thus remove conversion from the target and obtain one form of WTT with some extra axioms. For the second translation phase we remove these axioms by realising them using K, funext, and the necessary computation equalities, thus landing in WTT.

Homotopy. For those concerned by the use of axiom K to interpret equality, it is interesting to remark that the translation is done in a setting general enough that it can be instantiated to 2-level type theories [1]. By 2-level type theories we mean theories equipped with two equality types, one that is strict (with K and funext), and one that is the *usual* unconstrained equality meaning that in particular it can interpret the homotopy—or even univalent—equality (or path type).

Our translation can thus go from an *extensional* 2-level type theory (close in definition and usage to Voevodsky's Homotopy Type System [4]) to a *weak* 2-level type theory, that is a theory with a strict equality but no conversion.

Conclusion. We provide a translation from ETT to WTT that shows the former is conservative over the latter, thus proving that computation/conversion doesn't add logical power to type theory, including in a homotopy setting.

References

[1] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending homotopy type theory with strict equality. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, Marseille, France*, 2016.

[2] S. Boulier, M. Sozeau, N. Tabareau, and T. Winterhalter. Formalisation of ETT to ITT (and to WTT), 2019. Available at `https://github.com/TheoWinterhalter/ett-to-itt/tree/weak`.

[3] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages, POPL 2019, Cascais, Portugal*, 2019.

[4] Vladimir Voevodsky. A simple type system with two identity types. *Unpublished note*, 2013. `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf`.

[5] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. Eliminating Reflection from Type Theory. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Lisbonne, Portugal*, 2019.

Type-theoretic modalities for synthetic $(\infty, 1)$-categories

Ulrik Buchholtz and Jonathan Weinberger

TU Darmstadt, Darmstadt, GERMANY
{buchholtz,weinberger}@mathematik.tu-darmstadt.de

It is a challenge to find a convenient type-theoretic formulation of $(\infty, 1)$-category theory. In a recent approach inspired by the complete Segal space model of $(\infty, 1)$-categories, Riehl and Shulman [8] devise a type theory compatible with Voevodsky's univalence axiom yielding basic results of $(\infty, 1)$-category theory.

We present a variation of Riehl and Shulman's theory, extended by certain modalities in order to derive new results in this framework. Some of these modalities first appeared in connection with manifold-like cohesion, cf. [10]. As an example, we give an actual embedding of categories

$$\mathbf{y}_A : A \hookrightarrow (A^{\mathrm{op}} \to \mathrm{Space}), \tag{1}$$

illustrating two new features: the opposite modality $(-)^{\mathrm{op}}$, and a categorical universe of spaces and maps, Space.

Modalities for (truncated) simplicial spaces The intended semantics for Riehl–Shulman is in the $(\infty, 1)$-topos of simplicial spaces, $\mathrm{PSh}_\infty(\mathbb{\Delta})$, which is cohesive over the $(\infty, 1)$-topos of spaces, $\infty\mathrm{Gpd}$. Concretely, the theory is modeled in bisimplicial sets endowed with the Reedy model structure acting as type-theoretic model structure.

It is useful to consider as well the (non-standard) semantics in truncated simplicial spaces. These are the $(\infty, 1)$-toposes $\mathrm{PSh}_\infty(\mathbb{\Delta}_{\leq n})$ corresponding to the full subcategories $\mathbb{\Delta}_{\leq n} \subseteq \mathbb{\Delta}$ on the k-simplices with $k \leq n$. In particular, $\mathrm{PSh}_\infty(\mathbb{\Delta}_{\leq 1})$ is the $(\infty, 1)$-topos of reflexive graphs. Contrary to simplicial spaces, the $(\infty, 1)$-toposes of truncated simplicial spaces are not cohesive over $\infty\mathrm{Gpd}$, but they still model the discrete and codiscrete modalities of [10], because they are ∞-connected and ∞-local $(\infty, 1)$-toposes. They fail to be cohesive because the left-most adjoint Π does not preserve binary products.

We show that modal discreteness, *i.e.* being modal with respect to the discrete modality, does *not* in general coincide with *categorical discreteness*, by which we mean the internal notion of discreteness used in Riehl–Shulman's type theory (*i.e.* invertibility of all directed arrows in a type). This contrasts with the situation in simplicial spaces [8, Rem. 7.5]. The following leverages computations previously done in [2].

Theorem 1. *In $\mathrm{PSh}_\infty(\mathbb{\Delta}_{\leq 1})$ categorical discreteness is not equivalent to modal discreteness.*

We furthermore provide a characterization of the codiscrete modality of $\mathrm{PSh}_\infty(\mathbb{\Delta}_{\leq 1})$ as an accessible modality, cf. [9], as follows:

Theorem 2. *The codiscrete reflection in 1-truncated simplicial spaces is given by nullification of the fibers of the inclusion $1 + 1 \hookrightarrow \Delta^1$.*

In simplicial spaces, the shape modality computes the localization of an $(\infty, 1)$-category at all of its morphisms. The significance of this arises in its application as the group completion operation when applied to $(\infty, 1)$-categories with a pointed, connected space of objects.

The flat and sharp modalities both represent the core of an $(\infty, 1)$-category: For any type, there is a codiscrete reflection, and for crisp types, there is a discrete coreflection. In addition to the modalities from cohesion, we introduce another modality for the opposite type as previously announced in [1], coming from the order-reversing involution of $\triangle$.

Defining the opposite modality in a fruitful way turns out to be somewhat subtle. Using the modalities not only calls for additional definitional equalities, but one also requires a generalization of the preexisting *extension types* from the Riehl–Shulman framework, e.g. when defining $\hom_{A^{\mathrm{op}}}$ uniformly. Overall, we give an extension of Riehl–Shulman type theory extended with cohesive modalities, the opposite type modality, and a notion of generalized extension types.

Categorical universes Our desired application (1) uses the universe Space. This can be defined internally by embedding simplicial spaces in cubical spaces (modeled on the full subcategory of posets on the powers of $\{0 < 1\}$). There we can leverage the fact that Δ^1 is a tiny object, and then use the technique of [6] to define the universe Space as announced in [3].

To derive (1) from the version of the Yoneda embedding in [8] we also need to postulate the directed univalence axiom for Space. This holds in the intended semantics [4], and it can plausibly be given a constructive meaning by reusing the glue types of [5] in a new way [7].

From [3] we have further universes, including Dist[Space], Cat, and Dist[Cat]. We speculate on how to formulate the corresponding directed univalence axioms, which is not obvious because just looking at the 1-simplices is not sufficient in the case of distributors.

We do not claim any meta-theoretical properties (canonicity, decidable equality, etc.) of our extended type theory. However, by recent work of [11] one would be able to get universes à la Russell in (a type-theoretic model category presenting) cubical spaces.

References

[1] Ulrik Buchholtz and Jonathan Weinberger, *A mode theory for a type theory of cubical and simplicial types*, 2018, Presentation at EUTYPES Working group meeting, Aarhus.

[2] ______, *(Truncated) simplicial models of type theory*, 2018, Presentation at Workshop on Homotopy Type Theory/Univalent Foundations, Oxford.

[3] ______, *Universes in a type theory for synthetic ∞-category theory*, 2018, Presentation at EUTYPES Working group meeting, Aarhus.

[4] Evan Cavallo, Emily Riehl, and Christian Sattler, *On the directed univalence axiom*, 2018, Talk at AMS Special Session on Homotopy Type Theory, Joint Mathematics Meething, San Diego.

[5] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg, *Cubical type theory: a constructive interpretation of the univalence axiom*, 21st International Conference on Types for Proofs and Programs (TYPES 2015), LIPIcs. Leibniz Int. Proc. Inform., Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2018.

[6] Dan Licata, Ian Orton, Andy Pitts, and Bas Spitters, *Internal Universes in Models of Homotopy Type Theory*, ArXiv e-prints (2018).

[7] Dan Licata and Matthew Weaver, *Directed univalence in bicubical directed type theory*, 2018, Presentation at MURI Meeting, Pittsburgh.

[8] Emily Riehl and Michael Shulman, *A type theory for synthetic ∞-categories*, Higher Structures **1** (2017), no. 1, 147–224.

[9] Egbert Rijke, Michael Shulman, and Bas Spitters, *Modalities in homotopy type theory*, ArXiv e-prints (2017), arXiv:1706.07526.

[10] Michael Shulman, *Brouwer's fixed-point theorem in real-cohesive homotopy type theory*, Mathematical Structures in Computer Science **28** (2018), no. 6, 856941.

[11] Michael Shulman, *All $(\infty, 1)$-toposes have strict univalent universes*, arXiv e-prints (2019), arXiv:1904.07004.

Quillen bifibrations and the Reedy construction

Pierre Cagne (joint work with P.-A. Melliès)

IRIF, Université Paris Diderot – Paris 7
`cagne@irif.fr`

The Grothendieck construction is an ubiquitous tool that takes as input a (pseudo) functor $P : \mathcal{B}^{\mathrm{op}} \longrightarrow \mathsf{Cat}$ valued in the 2-category of (small) categories and produces a Grothendieck fibration $\mathcal{E}(P) \longrightarrow \mathcal{B}$. Even better, every Grothendieck fibration comes from such a pseudo functor, in an essentially unique way. If we restrict our attention to the pseudo functors P such that $P(u)$ has a left adjoint for every map u of $\mathcal{B}$, then the previous correspondence restricts to Grothendieck bifibrations. To fix notations, starting from a Grothendieck bifibration $p : \mathcal{E} \longrightarrow \mathcal{B}$, the (essentially unique) pseudo functor it comes from will be denoted by:

$$A \mapsto \mathcal{E}_A, \quad (u : A \longrightarrow B) \mapsto (u^* : \mathcal{E}_B \longrightarrow \mathcal{E}_A)$$

Whenever u^* has a left adjoint, denote it $u_!$. Most type-theoretic constructions find a natural semantics in such structures, first explored by Lawvere under the name “hyperdoctrines”. Extensional identity types *à la* Martin-Löf for example are easily described as $\delta_!(\top_A) \in \mathcal{E}_{A\times A}$ where $\top_A$ is terminal in $\mathcal{E}_A$ and $\delta_A : A \longrightarrow A \times A$ is the diagonal of A in $\mathcal{B}$. **Intentional** identity types however are not quite well encompassed by this framework. The lack of *reflection rule* for such identity types involves weaker structures to interpret them. With this incentive in mind, we design in this work an homotopical version of the Grothendieck construction that allows to equip $\mathcal{E}(P)$ with a Quillen model structure whenever the pseudo functor $P : \mathcal{B}^{\mathrm{op}} \longrightarrow \mathsf{Cat}$ starts from a model category and is valued in model categories in such a way that $P(u)$ is a Quillen right adjoint for every map u of $\mathcal{B}$, provided that P behaves nicely enough according to this homotopical content.

More precisely, let us define a *Quillen bifibration* as a Grothendieck bifibration $p : \mathcal{E} \longrightarrow \mathcal{B}$ with model structures on both $\mathcal{E}$ and $\mathcal{B}$ such that

- p preserves fibrations, cofibrations and weak equivalences,
- for each object A of $\mathcal{B}$, the morphisms of the fiber $\mathcal{E}_A$ that are fibrations, cofibrations and weak equivalences of $\mathcal{E}$ forms a model structure on $\mathcal{E}_A$.

Then we establish the following:

Theorem. *Let $p : \mathcal{E} \longrightarrow \mathcal{B}$ be a Grothendieck bifibration, such that $\mathcal{B}$ and each of the fibers $\mathcal{E}_A$, $A \in \mathcal{B}$ have model structures. Provided that the adjunctions $(u_!, u^*)$ are Quillen for all maps u of $\mathcal{B}$, the functor p is a Quillen bifibration if and only if the following conditions are met:*

(hCon) *the functor $u_!$ preserves and reflects weak equivalences whenever u is an acyclic cofibration; dually the functor v^* preserves and reflects weak equivalences whenever v is an acyclic fibration,*

(hBC) *for every commutative square of $\mathcal{B}$ as follows*

$$\begin{tikzcd} A \arrow[r, "v"] \arrow[d, "u'"'] & C \arrow[d, "u"] \\ C' \arrow[r, "v'"'] & B \end{tikzcd}$$

with u, u' acyclic cofibrations and v, v' acyclic fibrations, the induced mate *natural transformation $(u')_! v^* \longrightarrow (v')^* u_!$ is pointwise a weak equivalence in the fiber $\mathcal{E}_{C'}$.*

A prominent illustration and a second source of inspiration is the reconstruction through this theorem of the Reedy model structure on the category $\mathrm{Fun}(\mathcal{R},\mathcal{M})$ of diagrams of shape $\mathcal{R}$ (a Reedy category) in a model category $\mathcal{M}$. In the initial (unpublished) work of Kan, inspired by Reedy's work on simplicial objects in a model category, the weak equivalences of this model structure are given pointwise and the fibrations and cofibrations are presented through the so-called *latching* and *matching* spaces. The usual proof goes through a transfinite induction on the degree of the objects in the Reedy category $\mathcal{R}$, and the induction step for successor cardinals is quite technical. The previous result on Quillen bifibrations allows for a simpler explanation of this successor step and the use of these latching and matching spaces: if one denotes $\mathcal{R}_\mu$ the full subcategory of $\mathcal{R}$ spanned by the objects of degree strictly less than μ, then the restriction functor

$$\mathrm{Fun}\left(\mathcal{R}_{\mu+1},\mathcal{M}\right) \longrightarrow \mathrm{Fun}\left(\mathcal{R}_{\mu},\mathcal{M}\right)$$

is a Grothendieck bifibration that meets the hypothesis of the theorem if we equip the base $\mathrm{Fun}\left(\mathcal{R}_{\mu},\mathcal{M}\right)$ with the Reedy model structure. The conclusion of the theorem gives a model structure on the total category $\mathrm{Fun}\left(\mathcal{R}_{\mu+1},\mathcal{M}\right)$, which turns out to be the Reedy model structure one level higher! The latching and matching spaces emerge naturally as initial and final objects of the fibers, and the (co)cartesian gaps of the form $X \longrightarrow Y \times_{M_rY} M_rX$ and $X \sqcup_{L_rX} L_rY \longrightarrow Y$ used in Kan's original definition of the (co)fibrations come naturally as factorizations of $X \longrightarrow Y$ in the fibers through the (co)cartesian morphisms. These remarks on the Reedy construction also apply to the various generalizations of Reedy categories and the associated constructions: Berger-Moerdijk's [BM11], Cisinski's [Cis06] and Shulman's [Shu15].

In this talk, I will precisely state the theorem and try to give a feeling of what makes it tick in order to shed light on the Reedy construction. As an opening I will sketch the surprising behavior of the homotopy localization of Quillen bifibrations.

References

[BM11] Clemens Berger and Ieke Moerdijk. On an extension of the notion of Reedy category. *Mathematische Zeitschrift*, 269(3-4):977–1004, 2011.

[Cis06] Denis-Charles Cisinski. *Les préfaisceaux comme modèles des types d'homotopie*. Société mathématique de France, 2006.

[Shu15] Michael Shulman. Reedy categories and their generalizations, July 2015. arxiv:1507.01065v1.

Categories with Families: Unityped, Simply Typed, Dependently Typed (Extended Abstract)

Simon Castellan, Pierre Clairambault, Peter Dybjer

An important part of categorical logic is to establish correspondences between languages (from logic) and categorical models. For example, in their book "Introduction to higher order categorical logic" [7] Lambek and Scott prove equivalences between typed lambda calculi and cartesian closed categories, between untyped lambda calculi and C-monoids, and between intuitionistic type theories and toposes. Lambek and Scott's intuitionistic type theories are intuitionistic versions of Church's simple theory of types, which should not be confused with Martin-Löf's intuitionistic type theories. Interestingly, in the preface of their book [7, p viii] Lambek and Scott express a desire to include a result concerning the latter too:

> We also claim that intuitionistic type theories and toposes are closely related, in as much as there is a pair of adjoint functors between their respective categories. This is worked out out in Part II. The relationship between Martin-Löf type theories and locally cartesian closed categories was established too recently (by Robert Seely) to be treated here.

Seely's seminal paper [9] claims to prove that a category of Martin-Löf type theories is equivalent to a category of locally cartesian closed categories (lcccs). However, his result relies on an interpretation of substitution as pullback, and these are only defined up to isomorphism. It is not clear how to choose pullbacks in such a way that the strict laws for substitution are satisfied. This coherence problem was identified and solved by Curien [4] and Hofmann [6] who provided alternative methods for interpreting Martin-Löf type theory in lcccs (see also [5]). By using Hofmann's interpretation Clairambault and Dybjer [3] could then show that there is an actual biequivalence of 2-categories.

In this paper [2] (see `arXiv:1904.00827, April 2019`) we ask ourselves what it would take to add the missing chapter on Martin-Löf type theory and its correspondence with lcccs to the book by Lambek and Scott.

First we would need to add some material to Part 0 in the book on "Introduction to category theory", including introductions to lcccs, 2-categories, bicategories, pseudofunctors, and biequivalences. But more profoundly, our biequivalence theorem differs from Lambek and Scott's (and Seely's) equivalence theorems in important respects, since we replace Seely's category of Martin-Löf theories by a 2-category of categories with families (cwfs), with extra structure for type formers $\mathrm{I}_{\mathrm{ext}}, \Sigma, \Pi$, and pseudo cwf-morphisms which preserve the structure up to isomorphism. Thus cwfs with extra structure replace Martin-Löf theories on the "syntactic" side of the biequivalence.

This style of presenting the correspondence between "syntax" and "semantics" for Martin-Löf's dependent type theory applies equally well to the simply typed lambda calculus and the untyped lambda calculus, provided we consider subcategories of simply typed cwfs (scwfs), where types do not depend on variables, and of untyped cwfs (ucwfs), where there is only one type. As for full cwfs we need to provide extra structure for modelling λ-abstraction and application in the untyped λ-calculus and for modelling type formers in the simply typed λ-calculus.

This suggests that we ought to rewrite Lambek and Scott's Part I "Cartesian closed categories and λ-calculus" in a way which harmonizes with our presentation of the biequivalence between Martin-Löf type theory and lcccs.

Categories wifh families (cwfs) model the most basic rules of dependent type theory, those which deal with substitution, context formation, assumption, and general reasoning about equality. A key feature of cwfs is that the definition can be unfolded to yield a generalized algebraic theory in Cartmell's sense [1]. As such it suggests a language of cwf-combinators which can be used for the construction of initial cwfs (with extra structure for modelling type formers).

We prove several correspondence theorems between "syntax" in the guise of a number of cwf-based notions and "semantics" in the guise of some basic notions from category theory. Some of our theorems require "contextuality", a notion introduced by Cartmell [1] for his contextual categories. Others require "democracy", a notion introduced by Clairambault and Dybjer for their biequivalence theorems. Moreover, our equivalence theorems require strict preservation of chosen cwf-structure, while our biequivalence theorems only require preservation of cwf-structure up to isomorphism. In this way we can relate a number of notions from categorical logic such as cartesian operads, Lawvere's algebraic theories, Obtułowicz' algebraic theories of type λ-$\beta\eta$ [8], categories with finite products and limits, cccs, and lcccs, to the corresponding cwf-based notions. In addition to this we discuss different constructions of initial ucwfs, scwfs and cwfs (with extra structure) with and without explicit substitutions.

The purpose of our work is not so much to prove new results, but to suggest a new way to organize basic correspondence theorems in categorical logic, where the ucwf-scwf-cwf-sequence provides a smooth progression of the categorical model theory of untyped, simply typed, and dependently typed λ-calculi. We will also highlight some of the subtleties which arise when relating syntactic and semantic notions. Another important feature is that the correspondences between logical theories and categorical notions are now split into two phases: (i) equivalences and biequivalences between cwf-based notions and basic categorical notions, and (ii) the constructions of initial cwf-based notions. This yields an "abstract syntax" perspective of formal systems, where specific formalisms for untyped, simply typed and dependently typed λ-calculi are instances of the respective isomorphism classes of initial cwf-based notions. This is particularly important for dependent types and Martin-Löf type theory, since different authors make different choices in the exact formulation of the syntax and inference rules. Being initial in the appropriate category of cwfs is a suitable correctness criterion for these formulations.

References

[1] John Cartmell. Generalized algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[2] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. `arXiv:1904.00827`, April 2019.

[3] Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. *Mathematical Structures in Computer Science*, 24(6), 2014.

[4] Pierre-Louis Curien. Substitution up to isomorphism. *Fundamenta Informaticae*, 19(1,2):51–86, 1993.

[5] Pierre-Louis Curien, Richard Garner, and Martin Hofmann. Revisiting the categorical interpretation of dependent type theory. *Theoretical Computer Science*, 546:99–119, 2014.

[6] Martin Hofmann. On the interpretation of type theory in locally cartesian closed categories. In Leszek Pacholski and Jerzy Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*. Springer, 1994.

[7] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Number 7 in Cambridge studies in advanced mathematics. Cambridge University Press, 1986.

[8] Adam Obtułowicz. Functorial semantics of the type free lambda-betaeta calculus. In *Foundations of Computation Theory*, pages 302–307, 1977.

[9] R. A. G. Seely. Locally cartesian closed categories and type theory. *Proceedings of the Cambridge Philosophical Society*, 95:33–48, 1984.

Internally Parametric Cubical Type Theory[*]

Evan Cavallo and Robert Harper

Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
`ecavallo@cs.cmu.edu`, `rwh@cs.cmu.edu`

A polymorphic function—one that takes a type as an argument—is said to be *parametric* when its behavior is uniform in that type argument [Str67]. Reynolds uncovered a precise formulation of this property, one that is satisfied by polymorphic functions in many type systems: a function is polymorphic when it acts on relations [Rey83]. For example, if F is a function of type $\forall X.\ X \to X$, then it is parametric when for any relation $R \subseteq A \times B$, if $R(a, b)$ holds, then so too does $R(F[A](a), F[B](b))$. Among the theorems that follow from parametricity are many naturality principles, popularly known as "theorems for free" [Wad89].

Reynolds' parametricity is *denotational*: to be more precise, a function is parametric when its denotation acts on semantic relations. To show that polymorphic functions in the typed λ-calculus are parametric, Reynolds defines a relational model of the system, the existence of which implies that the denotation of any term also acts on relations. Especially with the advent of dependent type theory, however, we know that many denotational concepts have operational equivalents. This raises a natural question: can parametricity be exposed *within* type theory?

That question has recently been explored to great effect. Bernardy, Jansson, and Paterson observe that in a dependent type system, the relational interpretations of types and terms can be defined within the same system [BJP10]. This is a partial internalization: the interpretation function itself is external. Full internalization is achieved by Bernardy and Moulin, who introduce a type former that computes relational interpretations [BM12]. They have simplified their initial theory by the use of so-called *colors* [BM13, BCM15], while Nuyts, Vezzosi, and Devriese have developed a related theory, using modalities to capture distinctions between variables used in parametric and ad-hoc positions [NVD17].

We extend internal parametricity to *cubical type theory*, a theory of proof-relevant equality [CCHM15, AFH18]. The integration of the two is abetted by strong similarities between them, asevidenced by cross-pollination in their historical development. Cubical type theory adds a context Ψ of *path dimensions*: a term $M \in A\ [\Psi, x]$ varying in a dimension x is a *path*, or proof-relevant equality, connecting endpoints $M\langle 0/x\rangle \in A\langle 0/x\rangle\ [\Psi]$ and $M\langle 1/x\rangle \in A\langle 1/x\rangle\ [\Psi]$ given by substitution with constants $0, 1$. Path equality enjoys strong extensionality properties, including function extensionality and Voevodsky's *univalence axiom*, which identifies paths between types with equivalences. In parallel, internal parametricity contributes a context Φ of *bridge dimensions* (the aforementioned colors): a bridge of types $A\ \mathsf{type}\ [\Phi, \boldsymbol{x} \mid \Psi]$ is a relation between its endpoints $A\langle \mathbf{0}/\boldsymbol{x}\rangle$ and $A\langle \mathbf{1}/\boldsymbol{x}\rangle$, while a term $M \in A\ [\Phi, \boldsymbol{x} \mid \Psi]$ is evidence that $M\langle \mathbf{0}/\boldsymbol{x}\rangle$ is related to $M\langle \mathbf{1}/\boldsymbol{x}\rangle$ by A. In place of function extensionality, we have a characterization of bridges at function type as functions from bridges to bridges; in place of univalence, the identification of bridges between types with type-valued relations, a principle we call *relativity*.

By extending internal parametricity to cubical type theory, we can consider the parametricity properties of *higher inductive types*: inductive types with higher-dimensional generators. For example, the *suspension* $\mathsf{susp}(A)$ of a type A is generated by two elements north, south and a path $\mathsf{merid}(a)$ from north to south for every $a : A$. Using internal parametricity, we show that the functions of type $(X{:}\mathcal{U}) \to \mathsf{susp}(X) \to \mathsf{susp}(X)$ (where $\mathcal{U}$ is a universe of types) are

[*]This abstract is based on work supported by Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

characterized by where they send north and south: such a function is either the identity, constant north, constant south, or the function that sends north to south, south to north, and each merid path to its inverse. We believe that these techniques can be applied to establish algebraic properties of higher inductive types, such as the symmetric monoidal structure on the smash product [vD18], which currently require highly technical proofs.

In addition to proving results specific to cubical type theory, we advance the practice of internal parametricity. We suggest that relativity characterizes bridges in inductive types by showing that this is indeed the case for booleans: there is an equivalence between bridges in bool and paths in bool. We single out types with such trivial bridge structure as *bridge-discrete*; the sub-universe of bridge-discrete types is closed under most type formers (excluding universes) and relativistic, in the sense that its bridges correspond to relations valued in bridge-discrete types. Bridge-discreteness plays the role of the *identity extension lemma* of standard parametricity, which we demonstrate by example by proving that paths in a bridge-discrete type are given by Leibniz equality. Using the fact that bool is bridge-discrete, we refute the law of the excluded middle for homotopy propositions (evoking [Uni13, Corollary 3.2.7]).

Details, including a computational semantics, can be found in our technical report [CH19].

References

[AFH18] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. Cartesian cubical computational type theory: Constructive reasoning with paths and equalities. In *CSL 2018, September 4-7, 2018, Birmingham, United Kingdom*, 2018.

[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.

[BJP10] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 345–356, 2010.

[BM12] Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 135–144, 2012.

[BM13] Jean-Philippe Bernardy and Guilhem Moulin. Type-theory in color. In *ICFP 2013, Boston, MA, USA - September 25 - 27, 2013*, pages 61–72, 2013.

[CCHM15] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, pages 5:1–5:34, 2015.

[CH19] Evan Cavallo and Robert Harper. Parametric cubical type theory. arXiv:1901.00489, January 2019.

[NVD17] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *PACMPL*, 1(ICFP):32:1–32:29, 2017.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[Str67] Christopher Strachey. *Fundamental Concepts in Programming Languages*. Lecture notes, International Summer School in Computer Programming, Copenhagen, 1967.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[vD18] Floris van Doorn. *On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory*. PhD thesis, Carnegie Mellon University, 2018.

[Wad89] Philip Wadler. Theorems for free! In *FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359, 1989.

How to Tame your Rewrite Rules

Jesper Cockx[1], Nicolas Tabareau[2], and Théo Winterhalter[2]

[1] Chalmers, Göteborg, Sweden
[2] Gallinette Project-Team, Inria Nantes France

Dependently typed languages such as Coq and Agda can statically guarantee the correctness of our proofs and programs. We can extend these languages with new features using rewrite rules [5]. For example, exceptional type theory [8] introduces two new constructions $\mathsf{raise} : \forall A.\, A$ and $\mathsf{catch} : \forall P.\, P\ \mathsf{true} \to P\ \mathsf{false} \to P\ \mathsf{raise} \to \forall b.\, P\ b$ together with new rewrite rules such as $\mathsf{catch}_P\ pt\ pf\ pr\ \mathsf{raise} \rightarrowtriangle pr$. However in general, by adding a wrong rewrite rule, the language may lose any or all of its good properties like decidability of typechecking, canonicity, or even type safety. Moreover, these new rewrite rules may interact badly with other extensions.

We present a framework to add user-defined (higher-order and non-linear) rewrite rules to type theory in a *safe* and *modular* way. In particular, we provide checks to ensure type safety as well as decidability of conversion and thus type-checking, which in turn require checking the confluence and termination properties. We are currently working on extensions to both Agda and Coq to provide user-defined rewrite rules, where the user can pick their desired level of (un)safety by enabling or disabling individual checks (for confluence, termination, ...).

Ensuring subject reduction. In the current implementation of rewrite rules in Agda, 'bad' rewrite rules can destroy not only normalization and canonicity but also subject reduction:

- *Exploiting non-confluence:* Let $A : \mathsf{Set}$ with rewrite rules $A \rightarrowtriangle (\mathbb{N} \to \mathbb{N})$ and $A \rightarrowtriangle (\mathbb{N} \to \mathsf{Bool})$, then $(\lambda(x : \mathbb{N}).\, x)\ 0 : \mathsf{Bool}$. But this reduces to 0 which does not have type Bool.
- *Rewriting already defined symbols:* Let $\mathsf{Box}\ (A : \mathsf{Set}) : \mathsf{Set}$ be a datatype with a single constructor $\mathsf{box} : (x : A) \to \mathsf{Box}\ A$ and $\mathsf{unbox} : \mathsf{Box}\ A \to A$ defined by $\mathsf{unbox}\ (\mathsf{box}\ x) = x$. If we add a rewrite rule $\mathsf{Box}\ (\mathbb{N} \to \mathbb{N}) \rightarrowtriangle \mathsf{Box}\ (\mathbb{N} \to \mathsf{Bool})$, then we have $\mathsf{unbox}\ (\mathsf{box}\ (\lambda(x : \mathbb{N}).\, x))\ 0 : \mathsf{Bool}$ but this term again reduces to 0, which does not have type Bool.

The second example exploits the fact that $\mathsf{unbox}_A\ (\mathsf{box}_B\ x) \rightarrowtriangle x$ even when $A \neq B$. Here Agda implicitely assumes that Box is injective, enforcing the necessary conversion by typing. If we were to check for conversion in the reduction rule for unbox (as we do for user-defined rewrite rules) evaluation would be stuck but subject reduction would be preserved.

Both these examples break injectivity of Π types, which is a crucial lemma in most proofs of subject reduction. Hence we infer that we should check confluence of the rewrite rules, and we should only allow rewrite rules on 'fresh' (i.e. postulated) symbols. From these natural restrictions, we can derive injectivity of Π types and hence re-establish subject reduction.

Checking confluence and termination. Confluence and termination of higher-order rewrite rules are both known and well-studied problems (see for example [7] and [3]). However, checking termination usually requires confluence, and checking confluence usually requires termination. We propose to resolve this dilemma by fixing a deterministic *rewriting strategy* $\rightarrowtriangle_s \subseteq \rightarrowtriangle$, which is *complete* in the sense that whenever $u \rightarrowtriangle v$, there exists some w such that both $u \rightarrowtriangle_s^* w$ and $v \rightarrowtriangle_s^* w$. We can then check confluence and termination:

1. First, we check termination of $\rightarrowtriangle$. If it succeeds, we don't know yet that $\rightarrowtriangle$ is terminating (because the termination check assumes confluence), but we do know $\rightarrowtriangle_s$ is terminating (since it is included in $\rightarrowtriangle$ and confluent by construction).

2. Second, we check confluence of $\twoheadrightarrow$, using $\twoheadrightarrow_s$ for joining critical pairs. Because $\twoheadrightarrow_s \subseteq \twoheadrightarrow$ and $\twoheadrightarrow_s$ is complete, this check succeeds iff $\twoheadrightarrow$ is confluent.

3. Finally, we conclude that $\twoheadrightarrow$ is terminating, using point 1. and the confluence of $\twoheadrightarrow$.

Rewriting modulo an equational theory. In many proof assistants, conversion includes not just computation rules (e.g. β-reduction) but also type-directed rules (e.g. η-conversion). Hence we consider conversion up to an equational theory $\sim$. E.g. this relation can include η-rules for functions or records, or a definitionally proof-irrelevant universe of propositions [6].

If rewrite rules do not respect $\sim$, we run into trouble. For example, let $f : \forall A.\, (A \to A) \to \mathsf{Bool}$ with a rewrite rule $f_A\ (\lambda x.\, x) \twoheadrightarrow \mathsf{true}$ and consider the term $f_\top\ (\lambda x.\, \mathsf{tt})$. The argument $\lambda x.\, \mathsf{tt}$ is not of the form $\lambda x.\, x$, yet the rewrite rule should still apply since $\mathsf{tt} \sim x : \top$!

As such we must ensure that rewrite rules are well-behaved with respect to the equational theory: if $a \sim a' : A$ and $a \twoheadrightarrow b$, then there must exist b' such that[1] $b \sim b' : A$ and $a' \twoheadrightarrow^* b'$. With this property we are able to deduce that conversion between two terms $\Gamma \vdash t = u : A$ is equivalent to $t \twoheadrightarrow^* t'$ and $u \twoheadrightarrow^* u'$ and $\Gamma \vdash t' \sim u' : A$. This allows us to prove[2] injectivity of Π types and check confluence in the presence of a non-trivial equational theory.

Conclusion. User-defined rewrite rules allow you to extend the power of a dependently typed language on a much deeper level than normally allowed. We already mentioned exceptional type theory; other potential applications include adding new equations to neutral terms [1], defining quotient types [4] or higher inductive types, and implementing guarded type theory [2]. By having access to the necessary checks you can be confident that no important properties will break by accident, while you still have the option to ignore the checks when required by your application. Soon rewrite rules and the corresponding safety checks are coming to both Agda and Coq. We cannot wait to see what other uses you will come up with!

References

[1] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: A sound and complete decision procedure, formalized. In *Dependently-typed Programming*, 2013.

[2] Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *LICS '13*.

[3] Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. unpublished, 2019.

[4] Guillaume Brunerie. quotients.agda. `https://github.com/guillaumebrunerie/initiality/blob/reflection/quotients.agda`.

[5] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In *TYPES '16*.

[6] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *POPL '19*.

[7] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical computer science*, 1998.

[8] Pierre-Marie Pédrot and Nicolas Tabareau. Failure is Not an Option An Exceptional Type Theory. In *ESOP '18*.

[1] This does not assume that $b : A$, the type A is simply a hint to guide the equational theory.
[2] Further assuming $\Pi\ A'\ B' \sim \Pi\ C'\ D'$ is equivalent to $A' \sim C'$ and $B' \sim D'$

Frame type theory

Cyril COHEN[1], Assia MAHBOUBI[2], and Xavier MONTILLET[2]

[1] Université Côte d'Azur, Inria, France
[2] Inria, LS2N, France `firstname.lastname@inria.fr`

Writing modular programs in proof assistants is notoriously difficult. A significant literature[1] and implementation effort is devoted to the topic, with approaches ranging from adding new constructions to the underlying logic, to adding features to the proof assistant. However, all current options (including records, sections and modules [4, 2]) are unsatisfactory in one way or another. In this work (in progress), we aim at reconciling the pros of several options using frames. Figure 1 summarizes the pros and cons of each option as implemented in the Coq proof assistant [1], and the desired properties of frames.

	Sections	Modules	Records	Frames
First-class objects	No	No	Yes	Yes
Named application / Named currying	No	Yes / No	Yes / No	Yes
Minimal discharge / Field commutation	Yes	No	No	Yes
Import / Flattening	No	Yes	No	Yes
Refinement / Definitional singleton	No	Yes	No	Yes
Subtyping	No	Yes	No	No

Figure 1: Summary of pros and cons of each option

Conjecture 1. *There exists a calculus of* dependently typed frames *that satisfies the properties of figure 1, is confluent, is strongly-normalizing, has decidable type checking, and is conservative over CiC (extended with a definitional singleton).*

This talk will describe and discuss the properties of our candidate calculus.

Frames The central idea is to consider records where some fields do not have a value yet. We will call these generalized records frames, and will say that a field is a definition (resp. abstraction) if it has (resp. does not have) a value. Frames can also be thought of as a reification of the contexts of CiC, as presented in the Coq manual. For example, the frame $\tau \overset{\text{def}}{=} \{x := 1, \lambda y, z := x + y\}$ has two definitions x and z and one abstraction y. Projections are only allowed for definition fields that are not preceded by an abstraction field (that they depend on). For example $\tau.x \equiv 1$ but neither $\tau.y$ nor $\tau.z$ are defined.

From Coq to frames Vernacular commands of Coq (like Definition) can be thought of as acting on frames contexts, i.e. frames with a hole. For example, if we write τ for frames and p for Coq programs, the rules in Figure 2[2] define some mock-up operational semantics for Coq programs (where the box represents the hole of the frame context, and the program being evaluated is placed inside the hole).

$$\begin{array}{lcl}
\left\{\tau, \boxed{\mathsf{Definition}\, x := t.\ p}\right\} & \rhd & \left\{\tau, x := t, \boxed{p}\right\} \\
\left\{\tau, \boxed{\mathsf{Variable}\, x.\ p}\right\} & \rhd & \left\{\tau, \lambda x, \boxed{p}\right\} \\
\left\{\tau, \boxed{\mathsf{Module}\, M.\ p}\right\} & \rhd & \left\{\tau, M := \left\{\boxed{p}\right\}\right\} \\
\left\{\tau_1, M := \left\{\tau_2, \boxed{\mathsf{EndModule}\,.\ p}\right\}\right\} & \rhd & \left\{\tau_1, M := \{\tau_2\}, \boxed{p}\right\} \\
\left\{\tau_1, M := \{\tau_2\}, \tau_3, \boxed{\mathsf{Import}\, M.\ p}\right\} & \rhd & \left\{\tau_1, M := \{\tau_2\}, \tau_3, \tau_2, \boxed{p}\right\}
\end{array}$$

Figure 2: Mock-up operational semantics

[1]That we do not have the room to cite in due form.
[2]Where the last one is a special case for the sake of brevity.

First-class objects Just like records, frames should be first-class objects. This allows, for example, to define the n^{th} power of a monoid as its iterated product. It also allows to quantify over all real closed fields (with the carrier living in some fixed universe), for example to state that there is a quantifier elimination procedure.

Named application / Named currying / Partial instantiation Abstracting over several fields should be equivalent to abstracting over a record with those fields. In other words, it should be possible to give several arguments at once, with the names allowing to match abstractions and the corresponding arguments, and currying should be allowed:

$$\{\lambda x, \lambda y, z := x + y\}\,\{x := 0, y := 1\} \rhd^* \{x := 0, y := 1, z := 0 + 1\} \lhd^* (\{\lambda x, \lambda y, z := x + y\}\,\{x := 0\})\,\{y := 1\}$$

Minimal discharge / Field commutation Abstractions in frames should behave as abstractions in sections: If the value of some field x does not depend on the value of another field y (and does not depend either on fields that depend on y), then it should be possible to get the x projection of the frame without instantiating y. For example, $\{\lambda x, y := t, \dots\}.y \equiv t$ if x is not free in t. This allows to make each theorem proof depend only on the hypotheses it uses, without needing to make this set of hypotheses explicit.

More generally, fields that do not depend on each other should commute, including when one of the fields is a definition and the other one is an abstraction. For example, $\{\lambda x, y := t, \dots\} \equiv \{y := t, \lambda x, \dots\}$ if x is not free in t. Under these commutations, projections can be thought of as only being allowed for the first field (or more generally, fields that are the first field in a convertible frame).

Import / Flattening It should be possible to "import" a frame in another, just like for modules. This would allow to "reopen sections" by importing the corresponding frame. When compared to the use of records to represent sets equipped with some structure, this allows to "flatten" the structure without having to duplicate code (as when defining groups independently of monoids) or nest them (as when defining groups as a monoid with some extra structure). This "bundled vs unbundled" dilemma and related problems are described in [5, 3].

Refinement / Definitional singleton It should be possible to refine frames and their types. For example, the type of categories should be refinable to the type of monoids by forcing the type of objects to be `unit`, and the product of categories should be refinable to the product of monoids.

One could also define a ring as a monoid acting on an abelian group with the same carrier. This second example makes it clearer that we want the refined type to ensure that the two carriers are definitionally equal, and not just propositionally equal. In the context of the usual dependent types, this amounts to wanting a definitional singleton type $\mathrm{Sing}(t)$ inhabited by terms convertible to t, and such that a hypothesis $x : \mathrm{Sing}(t)$ in the context is understood as a definition $x := t$.

References

[1] The Coq Development Team. *The Coq Reference Manual, version 8.4*. Aug. 2012.

[2] Judicaël Courant. "MC_2 A module calculus for Pure Type Systems" (2007).

[3] François Garillot et al. "Packaging Mathematical Structures". 2009.

[4] Elie Soubiran. "Développement modulaire de théories et gestion de l'espace de nom pour l'assistant de preuve Coq. (Modular development of theories and name-space management for the Coq proof assistant)". PhD thesis. École Polytechnique, Palaiseau, France, 2010.

[5] Bas Spitters and Eelis van der Weegen. "Type classes for mathematics in type theory" (2011).

Sheaf Models of Univalent Type Theory

Thierry Coquand, Fabian Ruch

Göteborgs Universitet, Sweden
coquand@chalmers.se, fabian.ruch@cse.gu.se

Abstract

We adapt the technique of sheaf models used in topos theory to univalent type theory to obtain independence results for Markov's principle, countable choice, Brouwer's fan theorem, and LPO. We construct the models as inner models, refining the work in [6] and applying the theory of higher modalities [7].

1 Model construction

Following works such as [1, 5], various presheaf models of univalent type theory can be built in a constructive meta theory. In these models, we can consider a family $R\ c$ of (not necessarily fibrant) types over a fixed type C such that the functor $\square_c : X \mapsto (R\ c \to X)$ is an idempotent monad with unit $\eta_c : x \mapsto \lambda(\_ : R\ c).\ x$. It suffices, for instance, that $R\ c$ is a family of fibrant h-propositions [7]. We then consider, following [2], the higher inductive types $\square A$ (for arbitrary types A) and $\mathsf{N}_\square$ defined as follows.

$$\begin{aligned} \eta &: A \to \square A \\ \mathsf{g} &: \Pi(c : \mathsf{C})(\square_c(\square A) \to \square A) \\ \mathsf{s} &: \Pi(c : \mathsf{C})(x : \square A)(\mathsf{g}_c\ (\eta_c\ x) = x) \end{aligned}$$

$$\begin{aligned} \mathsf{zero} &: \mathsf{N}_\square \\ \mathsf{succ} &: \mathsf{N}_\square \to \mathsf{N}_\square \\ \mathsf{g} &: \Pi(c : \mathsf{C})(\square_c \mathsf{N}_\square \to \mathsf{N}_\square) \\ \mathsf{s} &: \Pi(c : \mathsf{C})(x : \square A)(\mathsf{g}_c\ (\eta_c\ x) = x) \end{aligned}$$

The constructor s ensures that each η_c has a left inverse and thus is an equivalence by idempotency.

Theorem 1.

- *$\square$ is a lex (left exact, finite-limit-preserving) modality [7, Definition 1.1, Theorem 3.1].*
- *In particular, $\square$ is the modal operator of a Σ-closed reflective subuniverse (and hence closed under dependent products, dependent sums, and identification types).*
- *The subtype $\mathsf{U}_\square \equiv \Sigma(X : \mathsf{U})\mathsf{isModal}(X)$ of U is $\square$-modal, where $\mathsf{isModal}(X)$ states that $\eta_c : X \to \square_c X$ is an equivalence for each $c : \mathsf{C}$.*
- *$\mathsf{U}_\square$ is a univalent universe of $\square$-modal types.*
- *The elimination principle $\mathsf{ext} : (\Pi(a : A)B(\eta\ a)) \to \Pi(x : \square A)B(x)$ for $\square A$ into a family B of $\square$-modal types satisfies the strict computation rule $(\mathsf{ext}\ f) \circ \eta \equiv f$.*
- *$\mathsf{N}_\square$ is $\square$-modal, and is a natural numbers type eliminating into $\square$-modal types with strict computation rules.*

Using this theorem we construct *inner* models inside the presheaf models along the lines of [6, Section 4.5], interpreting types as the $\square$-modal types. A new contribution w.r.t. [6] is that we also get models of inductive data types with *strict* computation rules, by mixing their constructors with the ones of the modality $\square$ (as shown above for the natural numbers type).

2 Applications

We can apply this model construction to obtain various independence and compatibility results for univalent type theory.

Theorem 2.

- *A countermodel for Markov's principle for decidable propositions P*
$$\mathsf{MP} \equiv (\neg\neg\Sigma(n:\mathsf{N})P(n)) \to \Sigma(n:\mathsf{N})P(n)$$
- *A countermodel for countable choice for families of sets A*
$$\mathsf{CC} \equiv (\Pi(n:\mathsf{N})\,\|A(n)\|) \to \|\Pi(n:\mathsf{N})A(n)\|$$
- *A model for Brouwer's fan theorem*
$$\mathsf{FT} \equiv (\Pi(\alpha:\mathsf{N}\to\mathsf{N}_2)\Sigma(n:\mathsf{N})A(\alpha,n)) \to \Sigma(M:\mathsf{N})\Pi(\alpha:\mathsf{N}\to\mathsf{N}_2)\Sigma(n:\mathsf{N}_{\leqslant M})A(\alpha,n)$$
- *A model for limited principle of omniscience*
$$\mathsf{LPO} \equiv (\Pi(n:\mathsf{N})P(n)) + \Sigma(n:\mathsf{N})\neg P(n)$$

The first two claims generalize to univalent type theory the results obtained in [3]. The model construction of the last claim is obtained by adapting [4] to univalent type theory.

A future research direction is to construct univalent sheaf models for the applications considered in the work of Wellen, like over rings equipped with the Zariski topology, for instance, where $\Box$ gets interpreted by the coreduction modality described in [8, Definition 2.3.2].

References

[1] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In T. Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[2] T. Coquand, S. Huber, and A. Mörtberg. On higher inductive types in cubical type theory. In A. Dawar and E. Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 255–264. ACM, 2018.

[3] T. Coquand, B. Mannaa, and F. Ruch. Stack semantics of type theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–11. IEEE Computer Society, 2017.

[4] T. Coquand and E. Palmgren. Intuitionistic choice and classical logic. *Arch. Math. Log.*, 39(1):53–74, 2000.

[5] I. Orton and A. M. Pitts. Axioms for modelling cubical type theory in a topos. In J. Talbot and L. Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, volume 62 of *LIPIcs*, pages 24:1–24:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[6] K. Quirin. *Lawvere-Tierney sheafification in Homotopy Type Theory. (Faisceautisation de Lawvere-Tierney en théorie des types homotopiques).* PhD thesis, École des mines de Nantes, France, 2016.

[7] E. Rijke, M. Shulman, and B. Spitters. Modalities in homotopy type theory. *CoRR*, abs/1706.07526, 2017.

[8] F. Wellen. *Formalizing Cartan Geometry in Modal Homotopy Type Theory.* PhD thesis, Karlsruher Instituts für Technologie, Germany, 2017.

Choreographies in Coq

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti[*]

Department of Mathematics and Computer Science, University of Southern Denmark
{lcf,fmontesi,peressotti}@imada.sdu.dk

Choreographic Programming is a paradigm for specifying concurrent systems based on message-passing where communications are written in an Alice-to-Bob notation. Every program (choreography) can then be mechanically translated into a distributed process-calculus implementation that is guaranteed to be bisimilar to the original choreography. Thanks to this methodology, such implementations are guaranteed never to suffer from mismatched communications. More generally, they cannot reach a deadlock state, since the original choreography language does not allow deadlocks.

Example 1. *The following choreography models a scenario where Alice* (a) *buys a book from a seller* (s) *routing the payment through her bank* (b)*.*

$$\begin{array}{l}\mathsf{a}.\mathit{title} \to \mathsf{s};\ \mathsf{s}.\mathit{price} \to \mathsf{a};\ \mathsf{s}.\mathit{price} \to \mathsf{b};\\ \mathit{if}\ \mathsf{b}.\mathit{ok}\ \mathit{then}\ \mathsf{b} \to \mathsf{s}[\mathit{ok}];\ \mathsf{b} \to \mathsf{a}[\mathit{ok}];\ \mathsf{s}.\mathit{book} \to \mathsf{a}\\ \qquad\quad \mathit{else}\ \mathsf{b} \to \mathsf{s}[\mathit{ko}];\ \mathsf{b} \to \mathsf{a}[\mathit{ko}]\end{array}$$

First, Alice sends the title of the book to the seller, which then quotes the price to both Alice and the bank. If the bank confirms the transaction, it sends an acknowledgement to both Alice and the seller, and the latter proceeds to send the book. Otherwise, the bank sends a cancellation to both parties.

The hallmark of Choreographic Programming is the EPP Theorem, which guarantees a precise operational correspondence between a choreography and its generated implementation (EndPoint Projection).

Example 2. *The previous choreography can be projected into the following distributed protocol.*

$$\begin{array}{rcl}\mathsf{a} & \triangleright & \mathsf{s}!\mathit{title};\ \mathsf{s}?;\ \mathsf{b}\&\{\mathit{ok}:\ \mathsf{s}?\mid \mathit{ko}:\}\\ \mathsf{b} & \triangleright & \mathsf{s}?;\ \mathit{if}\ \mathit{ok}\ \mathit{then}\ (\mathsf{s}[\mathit{ok}];\ \mathsf{a}[\mathit{ok}])\ \mathit{else}\ (\mathsf{s}[\mathit{ko}];\ \mathsf{a}[\mathit{ko}])\\ \mathsf{s} & \triangleright & \mathsf{a}?;\ \mathsf{a}!\mathit{price};\ \mathsf{b}!\mathit{price};\ \mathsf{b}\&\{\mathit{ok}:\ \mathsf{a}!\mathit{book}\mid \mathit{ko}:\}\end{array}$$

The protocol for Alice is thus: send a title to the seller and wait for a reply; then wait for either confirmation from the bank, in which case the seller will send the book, or cancellation, in which case the protocol ends. The protocol for the seller is similar. In turn, the bank initially waits for a message from the seller, and then decides whether to send confirmation or cancellation to both the seller and Alice.

In the examples above, the participants exchange two kinds of messages: data messages (e.g. $\mathsf{a}.\mathit{title} \to \mathsf{s}$) or signals (e.g. $\mathsf{b} \to \mathsf{s}[\mathit{ok}]$), which are exclusively meant to dictate control flow. The need for this distinction has to do with propagating local choices, in our example the decision by the bank on whether to authorize payment or not.

The EPP Theorem guarantees that the choregraphy in Example 1 behaves exactly as the three communicating processes in Example 2. However, the proof of this theorem even for

[*]This work was partially supported by the Independent Research Fund Denmark, grant no. DFF-7014-00041.

simple choreography languages is complex, due to the high number of cases that need to be considered and to the multitude of rules in the semantics of both choreography and process languages. Such proofs are known to be prone to errors when designed and checked by humans: a previous attempt to formalize a publication on a higher-order process calculus [3] turned up a number of problems in the original proofs [4].

Choreographic Programming is closely related to Multiparty Session Types, a typing discipline for concurrent programming that also guarantees desirable properties. The main difference between these two approaches is methodological: Multiparty Session Types work bottom-up, starting from an implementation and trying to find a type; Choreographic Programming works top-down, starting from a choreography (which can be thought of as a type with additional computational abilities and information on the data being communicated) and generating the implementation. It has recently been discovered that a significant number of published results in Multiparty Session Types were wrong, in the sense that not only did the published proofs contain errors, but also the stated results did not hold [5, Chapter 8.1]. Here again, the problem is the complexity of the proofs involved, both in terms of number of cases to be checked and technical complexity of checking each individual case.

In order to establish solid foundations for Choreographic Programming, we propose to formalize the core choreography calculus from [1] using the Coq theorem prover. This calculus was proposed originally as a minimal calculus that already embodies the characteristic features of Choreographic Programming. As such, it provides a good benchmark both to evaluate the feasibility of a full formalization of a model for Choreographic Programming and to verify its correctness by certified means. Moreover, this calculus already includes the major challenges that have to be dealt with in this theory, namely: finite sets and functions on finite sets; partial functions; syntactic binders.

Furthermore, [1] also includes a proof that this choreography model is Turing-complete. Formalizing this proof also requires formalizing Kleene's theory of partial recursive functions [2], which again deals with partiality and finite sets, but also poses some additional problems related to induction over dependent types.

Currently our formalization covers the fragment of the choreography language that does not include recursion (infinite behaviour). This fragment already requires treating finite sets and functions (as the semantics of choreographies is defined by means of a function assigning each process to the value it stores), as well as partial functions (even without recursion, projecting a choreography to a process implementation is not always possible). Furthermore, in this fragment we can already encode a subset of partial recursive functions. As such, this work is already illustrative of the challenges encountered and the solutions that can be put in place.

References

[1] Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. In Olga Kouchnarenko and Ramin Khosravi, editors, *FACS*, volume 10231 of *LNCS*, pages 17–35. Springer, 2017.

[2] S.C. Kleene. *Introduction to Metamathematics*, volume 1. North-Holland Publishing Co., 1952.

[3] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and decidability of higher-order process calculi. *Inf. Comput.*, 209(2):198–226, 2011.

[4] Petar Maksimovic and Alan Schmitt. HOCore in Coq. In Christian Urban and Xingyuan Zhang, editors, *ITP*, volume 9236 of *LNCS*, pages 278–293. Springer, 2015.

[5] Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *PACMPL*, 3(POPL):30:1–30:29, 2019.

Planar graphs in Homotopy Type Theory

Jonathan Prieto-Cubides[1] and Håkon Robbestad Gylterud[1]

Department of Informatics, University of Bergen, Bergen, Norway
{jonathan.cubides,hakon.gylterud}@uib.no

Abstract

We consider a characterization of planar graphs in Homotopy Type Theory (HoTT). Using basic concepts from HoTT, such as univalence, one can define a type of graphs, such that equality (in the sense of the identity type) between graphs coincides with isomorphism. For planarity, we take inspiration from topological graph theory, in particular, combinatorial embeddings of graphs into surfaces. A proof-assistant for dependent type theory with HoTT support (Agda) is used to verify the correctness of this work in progress.

Introduction. In Graph theory, a graph is *planar* when it can be embedded into the plane. There are many characterizations of planar graphs [2, 4], e.g. forbidden minors $K_{3,3}$ and K_5). For our definition, we are taking inspiration from Topological Graph Theory [1] where one works with combinatorial embeddings which represent embeddings of graphs into surfaces up to isotopy. A graph is planar if and only if it can be embedded into the sphere: if one has an embedding into the sphere, one can obtain an embedding into the plane, by puncturing the sphere and applying stereographic projection. We get a representation of planar graphs, up to isotopy, as a combinatorial embedding into the sphere, and keeping track of where the sphere was punctured.

In the following, we elaborate some of the concepts needed in order to formalize[1] the above characterization of planar graph in HoTT [3]. In particular, the notion of combinatorial embedding and which of those induce embeddings into the sphere. To start with, we fix a notion of graphs. In what follows, G and H are graphs and x, y or z are variables for nodes.

Graphs. We concern ourselves about *simple* and *undirected* graphs which can be formalized as the following type[2,3,4]:

$$\mathsf{Graph} :\equiv \sum_{\mathsf{N}:U} \sum_{\mathsf{E}:\mathsf{N}\to\mathsf{N}\to U} \mathsf{isSet}(\mathsf{N}) \times \prod_{x,y:\mathsf{N}} \mathsf{isProp}(\mathsf{E}\,x\,y) \times \prod_{x,y:\mathsf{N}} (\mathsf{E}\,x\,y \to \mathsf{E}\,y\,x).$$

Graph Homomorphisms. A *homomorphism* from G to H is a *pair* of functions (α, β) where $\alpha : \mathsf{N}_G \to \mathsf{N}_H$ acts on nodes and $\beta\,x\,y : \mathsf{E}_G\,x\,y \to \mathsf{E}_H(\alpha\,x)(\alpha\,y)$. When both functions are equivalences[5], such a map is called an *isomorphism*. As is typical of HoTT, the identity type on graphs is equivalent to the type of isomorphism between graphs[6].

Cyclic Orders. A *cyclic order* on a set A is a ternary relation R on A such that the binary relation $\mathsf{R}a$ is a total order for every $a : \mathsf{A}$ and for $a, b, c : \mathsf{A}$, if $a \neq b$, $\mathsf{R}abc$ implies $\mathsf{R}bca$.

Combinatorial Embeddings. A *combinatorial embedding* of a graph is a cyclic order on the star[7] of each node.

$$\mathsf{CombinatorialEmbedding}(G) :\equiv \prod_{x:\mathsf{N}_G} \mathsf{CyclicOrder}(\mathsf{Star}\,G\,x).$$

[1]Extra care is taken to choose types such that their identity types coincide with the natural notion of equivalences of the mathematical objects.

[2]For any type A, $\mathsf{isSet}(A) :\equiv \Pi_{x,y:A}\Pi_{p,q\,:\,x=y}\ (p = q)$.

[3]For any type A, $\mathsf{isProp}(A) :\equiv \Pi_{x,y:A}\ (x = y)$.

[4]A *node* is of type N_G and an *edge* between nodes x, y is of type $\mathsf{E}_G\,x\,y$.

[5]Equivalence of $\alpha : \mathsf{N}_G \to \mathsf{N}_H$ is bijection and for $\beta\,x\,y$ is bi-implication.

[6]The natural map $\Pi_{G,H:\mathsf{Graph}}\,(G = H) \to (G \cong H)$ is an equivalence.

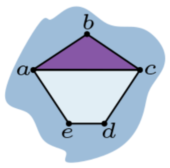
[7]$\mathsf{Star}_G(x) :\equiv \Sigma_{y:\mathsf{N}_G}\ \mathsf{E}_G\,y\,x$.

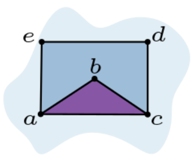
A combinatorial embedding defines an embedding of the graph into a closed surface but *which* surface is left implicit by the definition. However, the surface can be reconstructed from a combinatorial embedding by the notion of a face. We then recognize when the resulting surface is a sphere by using the fact the sphere is simply connected.

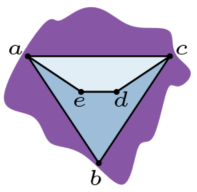
Cyclic Graphs. Identifying Fin_n with the set of $\{0, \cdots, n-1\}$, consider the function $S : \mathsf{Fin}_n \to \mathsf{Fin}_n$ which maps $(n-1 \mapsto 0,\ i \mapsto i+1)$. This function is an equivalence. From the function S, we construct the graph $\mathsf{C_n}$ for $n \geq 3$ with nodes Fin_n and edges from i to $S\ i$ for all $i : \mathsf{Fin}_n$. The type of *cyclic graphs* is the connected components of $(\mathsf{C_n}, S)$ in the type of all such structures[8]. A *cycle* in a graph G is cyclic graph H along with a homomorphism $H \to G$.

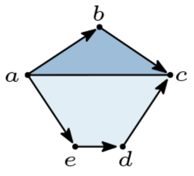
Corners. On the star x, a *corner* is a relation between two edges e_1,e_2 which satisfies there is no other edge in between. The corner is denoted by $e_1 \prec_{\mathsf{Star}_G(x)} e_2$.

Faces. A *face* is a cycle where all consecutive edges are corners and each corner occurs at most once. For example, the graph (I) below with the indicated combinatorial embedding has three faces with three, four, and five edges respectively.

(I) 

(II) 

(III) 

(IV) 

Spherical Graphs. A combinatorial embedding of a graph is *spherical* if any walk[9] w_1 can be obtained by *deforming along faces* any other walk w_2 with the same endpoints. For example in (IV), the walk $a - b - c$ can be deformed into the walk $a - e - d - c$ along the triangular face abc and the face $acde$.

Planar Graphs. As spherical combinatorial embeddings, (I) (II) and (III) are all equivalent. To distinguish these as embeddings into the plane, we can keep track of a face, designated as the *outer face*. We require for technical reasons that planar graphs are connected.

$$\mathsf{Planar}(G) :\equiv \sum_{e:\mathsf{CombinatorialEmbedding}_G} \mathsf{Spherical}\, G\, e \times \mathsf{Face}\, G\, e \times \mathsf{Connected}\ G.$$

Conclusions. The predicate Planar is not a proposition, in fact we expect to have elements representing all embeddings of the graph into the plane, identified up to isotopy. We would like to compare this approach with other characterizations, of planarity. While we here focus on planar graphs, it is also possible to consider graphs with rotation system as a way to specify any closed orientable surface. Constructing surfaces in this way, using higher inductive types, is another interesting line of investigation.

References

[1] Jonathan L Gross and Thomas W Tucker. *Topology Graph Theory.* 1987.

[2] Lars Noschinski. Formalizing Graph Theory and Planarity Certificates. 2015.

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

[4] M. Yamamoto, S. Nishizaki, M. Hagiya, and Y. Toda. Formalization of planar graphs. pages 369–384, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[8] $\mathsf{CyclicGraph} :\equiv \Sigma_{A:\mathsf{Graph}} \Sigma_{\varphi:A\to A} \Sigma_{n:\mathbb{N}} \parallel (A, \varphi) = (\mathsf{C_n}, S) \parallel$.

[9] A *walk* from x to y is a sequence of edges $e_1 : \mathsf{E}_G\, x\, x_1, \cdots, e_i : \mathsf{E}_G\, x_{i-1}\, x_i, e_i : \mathsf{E}_G\, x_i\, x_{i+1}, \cdots, e_n : \mathsf{E}_G\, x_k\, y$.

Verse - An EDSL in Coq for verified low-level cryptographic primitives

Abhishek Dang[1] and Piyush Kurur[2]

[1] Indian Institute of Technology Kanpur, India.
[2] Indian Institute of Technology Palakkad, India.

1 Introduction

Despite impressive work on code optimisation in modern compilers, cryptographic primitives are still being written in low-level languages like C and even assembly. While performance is an important consideration, the unpredictable nature of modern optimising compilers for high level languages leave primitives written in them vulnerable to various side channel attacks. However, in this bargain we have lost out on the safety, portability, and maintainability that comes naturally with a high-level language. We present *Verse* [4], a DSL embedded inside Coq for writing cryptographic primitives, which addresses some of these shortcomings.

2 The Design of Verse

Verse is a low-level language for writing cryptographic primitives albeit with a relatively high-level interface. Following in the footsteps of `qhasm` [2], the instruction set of Verse is simply a common notation for the arithmetic and bit-wise operations across architectures. However, unlike `qhasm`, it was possible to encode a lot of safety properties (including array index safety) into the inductive types that represent these instructions. We do this without compromising readability or ease of coding - the Coq notation system hides the arcane constructors of syntax elements of Verse arising out of the *correct by construction* strategy, and tactics fill in the proof objects that reside in, say, the array dereference constructor. In addition, a Verse programmer can generate Verse code using functions written in Gallina, the functional programming language underlying Coq. This meta-programming facility can be seen as assembler macros and is used to give many utility functions for coding patterns like looping, register caching etc. These features of Verse make the programming experience remarkably high-level. To the programmer, Verse appears to be a typed low-level language, with facility for defining and using assembler macros and strong compile time type safety. The interested reader can refer to the quick example in our repository[1] for some of these code features in action.

Finally, towards portability, we provide a framework to modularly add machine architectures to Verse. These specify the mostly one-to-one translation of Verse instructions to those of the machine and also abstract out some of the non-application code from Verse. This means, for instance, that while the user has complete control over the allocation of local variables to registers, he does not have to worry about parameter allocation by calling conventions or function preamble and cleanup for the local variables. *Verse* code can currently be compiled down to portable C and X86-64.

To demonstrate the feasibility of our approach, we have implemented some real world cryptographic code which can be found in our source code repository[2]. The resulting C code is now

[1]`https://github.com/raaz-crypto/verse-coq/blob/master/src/Verse/Tutorial.v`
[2]`https://github.com/raaz-crypto/verse-coq/tree/master/src/Verse/CryptoLib`

part of the Raaz[3] cryptographic library which will be released soon.

3 Functional properties

The primary motivation of embedding inside Coq, a proof assistant, was to prove functional properties of Verse code in the same environment where the code is written. The author's experience in developing Raaz informed our choice to limit focus to a simple instruction set - arithmetic and bit-wise operations - and a linear control flow. This design choice immunizes implementations against some timing attacks while not being too restrictive for the domain at hand. Further, these simplifications make for a simple semantics involving interpretation over a state machine. This was our approach towards semantic verification in our writeup [4]. However, this required understanding the internals of Verse to formulate and prove semantic properties, which compares unfavourably to the programming interface, where no internals are exposed. The parametric nature of the Verse AST and heavy use of the Coq sectioning mechanism in writing the code makes our job of giving a clean interface for proofs hard.

Our current approach is to provide the facility to add annotatations encoding program properties in Verse code itself. Extraction and presentation of proof obligations from annotated code is now very usable. We refer the reader to the annotated SHA2 code in our repository. The current tactics to extract out proof obligations from code work well with true bitvector code. We are working towards handling arithmetic proof obligations that come up with high-modulo arithmetic in elliptic curve primitives. It might bear remarking here that our aim is presentation of the proof obligations to the user in a palatable form, more so than to provide automated proofs for the obligations.

4 Conclusion

We hope to make a case for an embedded language as the approach towards narrow programming domains intent upon security and verification. This is in contrast to other work [1] [3]. Elsewhere, while not sharing our aim of providing a coding environment for general Cryptographic code, the Coq based approach [5] focuses on synthesis of verified fast implementations of high-modulo arithmetic.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823. ACM, 2017.

[2] Daniel J Bernstein. Writing high speed software, 2007.

[3] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *the 26th USENIX Security Symposium*, pages 917–934, 2017.

[4] Abhishek Dang and Piyush P Kurur. Verse: An EDSL for cryptographic primitives. In *the 20th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2018.

[5] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Systematic generation of fast elliptic curve cryptography implementations, 2018.

[3]https://github.com/raaz-crypto/raaz

Logipedia: a multi-system encyclopedia of formal proofs

Gilles Dowek and François Thiré

Inria and École normale supérieure de Paris-Saclay
LSV, 61, avenue du Président Wilson 94235 Cachan Cedex, France
`gilles.dowek@ens-paris-saclay.fr`
`francois.thire@lsv.fr`

Libraries of formal proofs are an important part of our mathematical heritage, but their usability and sustainability is poor. Indeed, each library is specific to a proof system, sometimes even to some version of this system. Thus, a library developed in one system cannot, in general, be used in another and when the system is no more maintained, the library may be lost. This impossibility of using a proof developed in one system in another has been noted for long and a remediation has been proposed: as we have empirical evidence that most of the formal proofs developed in one of these systems can also be developed in another, we can develop a standard language, in which these proofs can be translated, and then used in all systems supporting this standard.

Logipedia (`http://logipedia.science`) is an attempt to build such a multi-system on-line encyclopedia of formal proofs expressed in such as standard language. It is based on two main ideas: the use of a logical framework and of reverse mathematics.

Logical frameworks Different proof systems, such as Coq, Matita, HOL Light, Isabelle/HOL, PVS... implement variants of different *logical formalisms*: the Calculus of constructions, Simple type theory, Simple type theory with predicate subtyping... After several decades of research, we understand the relationship between these formalisms much better. But, to build an encyclopedia of formal proofs, we have been one step further and expressed all these formalisms as theories in a common *logical framework*.

The idea of using a logical framework, such as predicate logic, to express theories, such as geometry and set theory, goes back to Hilbert and Ackermann, but several other logical frameworks such as λ-*Prolog*, *Isabelle*, *the* $\lambda\Pi$-*calculus deduction modulo theory*, and *the* $\lambda\Pi$-*calculus modulo theory* [3, 5], have been proposed to solve some issues of predicate logic. We have used the *the* $\lambda\Pi$-*calculus modulo theory*, implemented in the system Dedukti [2]. With respect to predicate logic this logical framework allows

- binders (like λ-*Prolog*, *Isabelle*, and *the* $\lambda\Pi$-*calculus*)
- proofs as λ-terms (like *the* $\lambda\Pi$-*calculus*)
- an arbitrary definitional / computational equality (like *deduction modulo theory*)
- arbitrary connectives and quantifiers, in particular a mix of constructive classical ones (like ecumenical logics [6]).

It permits to express Simple type theory with 8 symbol declarations and 3 rewrite rules and the Calculus of constructions with 9 symbol declarations and 4 rewrite rules [4].

Expressing the theory implemented in HOL Light and Matita theories D[HOLL] and D[Mat] in Dedukti allows to translate proofs developed in these systems to these theories and back.

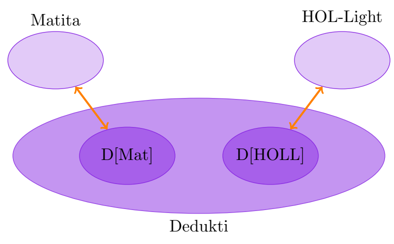


Figure 1: Logical Framework

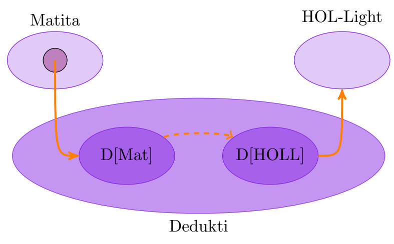


Figure 2: Reverse mathematics

Reverse mathematics Expressing various formalisms as theories in the same logical framework permits to compare them. For instance comparing the expression of Simple type theory and of the Calculus of constructions in Dedukti [4], we notice that there are only three differences the symbol *arrow* that permits to build functional types is dependent in the Calculus of constructions, and the same holds for the implication $\Rightarrow$. Then the Calculus of constructions has one extra symbol π to build types for functions mapping proofs to terms and Simple type theory does not. Thus all proofs expressed in D[HOLL] can be translated to D[Mat].

Conversely, we can identify a subset of the proofs of D[Mat], that do not use the dependency of the symbols *arrow* and $\Rightarrow$ and do not use the symbol π, and can be translated to D[HOLL]. Composing these translation we obtain a partial function mapping Matita proofs to HOL Light proofs.

Instead of keeping proofs in various libraries such that that of HOL Light or Matita, and use translators, we can as well build a single multi-system encyclopedia such as Logipedia and use them directly when developing a proof in one system or the other.

Empirical results As a proof of concept, we have translated to D[Mat] the arithmetic library of Matita [1] up to Fermat's little theorem (around 300 lemmas). We have then translated it to D[HOLL] and exported it to HOL Light, Isabelle/HOL, Coq, Lean, PVS, and, of course, Matita.

The proofs are available at `http://logipedia.science`.

References

[1] Ali Assaf. *A framework for defining computational higher-order logics. (Un cadre de définition de logiques calculatoires d'ordre supérieur).* PhD thesis, École Polytechnique, Palaiseau, France, 2015.

[2] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the lambda-Pi-calculus modulo theory. Submitted to publication.

[3] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Proceedings*, 2007.

[4] Gilles Dowek. Analyzing individual proofs as the basis of interoperability between proof systems. In *Proceedings of the Fifth Workshop on Proof eXchange for Theorem Proving, PxTP 2017.*, 2017.

[5] Bengt Nordström, Kent Petersson, and Jan M. Smith. Martin-Löf's type theory. In *Handbook of Logic in Computer Science*. 2000.

[6] Luiz Carlos Pereira and Ricardo Rodriguez. Normalization, soundness and completeness for the propositional fragment of Prawitz' ecumenical system. *Revista Portuguesa de Filosofia*, 2017.

Compact, totally separated and well-ordered types in univalent mathematics

Martín Hötzel Escardó

School of Computer Science, University of Birmingham, UK
m.escardo@cs.bham.ac.uk

Our univalent type theory. We work with an intensional Martin-Löf type theory with an empty type $\mathbb{0}$, a one element type $\mathbb{1}$, a type $\mathbb{2}$ with points 0 and 1, a type $\mathbb{N}$ of natural numbers, and type formers $+$ (disjoint sum), Π (product), Σ (sum), W types, and Id (identity type), and a hierarchy of type universes closed under them, ranged over by $\mathcal{U}, \mathcal{V}, \mathcal{W}$. On top of that we add Voevodsky's univalence axiom and a propositional truncation axiom.

A formal version of the development discussed here is available at our github repository TypeTopology, in Agda with the option –without-K. We are considering porting this to cubical Agda, so that no axioms are used and our results get a computational interpretation.

By a *proposition* we mean a type with at most one element (any two of its elements are equal in the sense of the identity type). The *existential quantification* symbol $\exists$ denotes the propositional truncation of Σ. We denote the identity type $\mathrm{Id}\, X\, x\, y$ by $x = y$ with X elided. We assume the notation and terminology of the HoTT Book unless otherwise stated.

Compact types. We consider three notions of exhaustively searchable type. We say that a type X is *compact*, or sometimes Σ-*compact* for emphasis, if the type $\Sigma(x : X), p\, x = 0$ is decidable for every $p : X \to \mathbb{2}$, so that we can decide whether p has a root. We also consider two successively weaker notions, namely $\exists$-*compactness* (it is decidable whether there is an unspecified root) and Π-*compactness* (it is decidable whether all points of X are roots), obtained by replacing Σ by $\exists$ and Π in the definition of compactness.

For the model of simple types consisting of Kleene–Kreisel spaces, these notions of compactness agree and coincide with topological compactness under classical logic, but we reason constructively here, meaning that we don't invoke (univalent) choice or excluded middle.

Finite types of the form $\mathbb{1} + \mathbb{1} + \cdots + \mathbb{1}$ are clearly compact. The compactness of $\mathbb{N}$ is LPO (limited principle of omniscience), which happens to be equivalent to its $\exists$-compactness, and its Π-compactness is equivalent to WLPO (weak LPO), and hence all forms of compactness for $\mathbb{N}$ are not provable or disprovable in our classically/constructively-neutral foundation.

An example of an *infinite* compact type is that of conatural numbers, $\mathbb{N}_\infty$, also known as the *generic convergent sequence* (this was presented in Types'2011 in Bergen). This type, the final coalgebra of $- + \mathbb{1}$, is not directly available in our type theory, but can be constructed as the type of decreasing infinite binary sequences.

We are able to construct plenty of *infinite* compact types, and it turns out they all can be equipped with well-orders making them into ordinals.

Ordinals. An ordinal is a type X equipped with a proposition-valued binary relation $- < - : X \to X \to \mathcal{U}$ which is transitive, well-founded (satisfies transfinite induction), and extensional (any two elements with the same predecessors are equal). The HoTT Book additionally requires the type X to be a set, but we show that this follows automatically from extensionality. For example, the types of natural and conatural numbers are ordinals. By univalence, the type of ordinals in a universe is itself an ordinal in the next universe, and in particular is a set.

Addition is implemented by the type former $- + -$, and multiplication by the type former Σ with the lexicographic order. The compact ordinals we construct are, moreover, *order-compact* in the sense that a minimal element of $\Sigma(x : X), p\,x = 0$ is found, or else we are told that this type is empty. Additionally, we have a selection function of type $(X \to \mathbb{2}) \to X$ which gives the infimum of the set of roots of any $p : X \to \mathbb{2}$, and in particular our compact ordinals have a top element by considering $p = \lambda x.1$.

Discrete types. We say that a type is *discrete* if it has decidable equality.

Totally separated types. It may happen that a non-trivial type has no nonconstant function into the type $\mathbb{2}$ of booleans so that it is trivially compact. For example, this would be the case for a type of real numbers under Brouwerian continuity axioms. Under such axioms, such types are compact, but in a uninteresting way. We say that a type is *totally separated*, again borrowing a terminology from topology, if the functions into the booleans separate the points, in the sense that any two points that satisfy the same boolean-valued predicates are equal. This can be seen as a boolean-valued Leibniz principle. Such a type is automatically a set, or a 0-groupoid, in the sense of univalent mathematics. We construct a totally separated reflection for any type, and show that a type is compact, in any of the three senses, if and only if its totally separated reflection is compact in the same sense.

Interplay between the notions. We show that compact types, totally separated types, discrete types and function types interact in very much the same way as their topological counterparts, where arbitrary functions in type theory play the role of continuous maps in topology, *without* assuming Brouwerian continuity axioms. For instance, if the types $X \to Y$ and Y are discrete then X is Π-compact, and if $X \to Y$ is Π-compact, and X is totally separated and Y is discrete, then X is discrete, too. The simple types are all totally separated, which agrees with the situation with Kleene–Kreisel spaces, but it is easy to construct types which fail to be totally separated (e.g. the homotopical circle) or whose total separatedness gives a constructive taboo (e.g. $\Sigma(x : \mathbb{N}_\infty), x = \infty \to \mathbb{2}$, where we get two copies of the point ∞).

Notation for discrete and compact ordinals. We define infinitary ordinal codes, or expression trees, similar to the so-called Brouwer ordinals, including one, addition, multiplication, and countable sum with an added top point.

We interpret these trees in two ways, getting discrete and compact ordinals respectively. In both cases, addition and multiplication nodes are interpreted as ordinal addition and multiplication. But in the countable sum with a top point, the top point is added with $- + \mathbb{1}$ in one case, and so is isolated, and by a limit-point construction in the other case (given our sequence $\mathbb{N} \to \mathcal{U}$ of types, we extend it to a family $\mathbb{N}_\infty \to \mathcal{U}$ so that it maps ∞ to a singleton type, by a certain universe injectivity construction, and then take its sum).

We denote the above interpretations of ordinal notations ν by Δ_ν and K_ν. The types in the image of Δ are discrete and retracts of $\mathbb{N}$, and those in the image of K are compact, totally separated and retracts of the Cantor type $\mathbb{N} \to \mathbb{2}$. Moreover, there is an order preserving and reflecting embedding $\Delta_\nu \to K_\nu$, which is an isomorphism if and only if LPO holds, but whose image always has empty complement for all ordinal notations ν. An example of such a situation is the evident embedding $\mathbb{N} + \mathbb{1} \to \mathbb{N}_\infty$ (this inclusion is merely a monomorphism, rather than a topological embedding, in topological models – the word *embedding* in univalent mathematics refers to the appropriate notion for ∞-groupoids, which in this example are 0-groupoids). By transfinite iteration of the countable sum, one can get rather large compact ordinals.

Deciding several concepts of finiteness for simple types

José Espírito Santo[1], Ralph Matthes[2], and Luís Pinto[1]

[1] Centro de Matemática, Universidade do Minho, Portugal
[2] Institut de Recherche en Informatique de Toulouse (IRIT), CNRS and Univ. of Toulouse, France

Proofs in propositional logic correspond to λ-terms with simple types – this is the Curry-Howard correspondence. The search for proofs of a given formula corresponds to the search for λ-terms of that type, since formulas are seen as types. Finite successful runs of the search correspond to the construction of inhabitants of that type, but proof search is more than the set of its finite successful outcomes. In our previous work (started in [1] and briefly described in [2]), we developed a representation of the search space for locally correct, bottom-up applications of the proof rules, not limiting the representation to the construction of (finite) inhabitants. The data structure we propose is an extension of the λ-calculus (more precisely: of a fragment to describe long normal forms) to a coinductive structure that also allows to express choice points in the search process. The elements of that structure are called forests, and individual inductive or even coinductive λ-terms can be seen as members of such a forest. In particular, for a given sequent $\sigma = \Gamma \Rightarrow A$, consisting of a context Γ (a set of declarations of types for variables) and a type A, we can define the canonical search space associated with σ as the forest S_σ. The finite members of S_σ are precisely the inhabitants of σ. More generally, the members of S_σ, whether finite or not, represent successful runs of the search procedure (that is, runs which did not face the impossibility of applying a proof rule); they also represent solutions of the proof search problem posed by the sequent.

The forests form a coinductive datatype and are therefore not necessarily finitely described objects. The finitary counterpart to forests is a λ-calculus with inductively defined terms (called finitary forests) that also has the means of expressing choice points and that comes with a formal fixed-point operator, based on fixed-point variables that are typed with sequents. There is a natural interpretation of finitary forests as forests, which henceforth allows to check whether finitary forests put forward are indeed solutions to problems specified semantically (in terms of forests). The first instance of that methodology is seen in the definition of the finitary forest F_σ for a given sequent σ (the "finitary representation of σ") whose interpretation is precisely S_σ. The termination of the recursive definition of F_σ exploits the subformula property of (minimal) propositional logic.[1]

The inhabitation problem – is there an inhabitant for a given sequent σ? – is the first problem we dealt with in this methodology, and this is sketched in [2] and developed in [4]. With a development that mostly mirrors the steps for inhabitation, we can also decide type finiteness – are there only finitely many inhabitants for a given sequent σ? Having only finitely many finite members of a forest is inductively characterized by a predicate finfin on forests, hence by duality its complement is a coinductive predicate inffin (the latter predicate is needed for coinductive reasoning). We have type finiteness of σ iff $\mathsf{finfin}(S_\sigma)$. On the level of finitary forests, we introduce a parameterized recursive predicate, FF_P, where the parameter P is a set of "good" sequents. Importantly, in one of the clauses of its definition, we need in the premise the parameterized predicate used for inhabitation, with its parameter set so as to exploit our algorithm for deciding inhabitation. The theorem on type finiteness characterizes type finiteness of σ by the truth of $\mathsf{FF}_\emptyset(F_\sigma)$.

[1]The results reported so far can be consulted on the arXiv [3], including an extensive discussion of the decontraction operation that is needed for the interpretation of finitary forests but not mentioned in this abstract.

In a similar spirit, one can obtain soundness of a simple counting function on finitary forests that yields the number of inhabitants if there are only finitely many. Two simple properties of finitary forests – having no occurrence of fixed-point variables and having no choice points with at least two options, respectively, can be proven for F_σ if σ is positively non-duplicated resp. negatively non-duplicated (under an extra proviso). This reproves (and factorizes) generalized coherence theorems (that traditionally give sufficient conditions for having uniqueness of proofs).

We now come to as of now unpublished material [5]. Type finiteness was taken above to mean that the sequent only has finitely many inhabitants. But, as soon as one grants the status of "member" of a sequent σ to any member of the forest S_σ, other natural concepts of finiteness are possible. One is the finiteness of any member of σ. This concept can be related to the finiteness of the search space, in the spirit of König's lemma (because the search space is a kind of tree whose branches are runs). So, finiteness of the search space is another concept of finiteness. Our main result is that all these concepts of finiteness are decidable, and so is the property of absence of members (finite or otherwise), which is an extreme form of unprovability.

An interesting observation is that all three mentioned concepts of finiteness of a sequent σ (type finiteness, finiteness of all members, finiteness of the search space) are instances of a generic finiteness predicate $\mathsf{fin}^\Pi(S_\sigma)$, the instances being determined by setting the parameter Π, which is a predicate on forests (e. g., finfin is obtained with Π set to the forests that have finite members). This allows an easy ordering of these three instances, leading to the conclusion that a sequent has only finitely many inhabitants as soon as all of its members are finite.

Parameterization also allows a single proof of decidability, capturing the three referred instances. The proof follows our methodology of obtaining an equivalent predicate $\mathsf{FIN}^\Pi(F_\sigma)$. In the proof of decidability, we require that Π is decidable, when composed with a certain alternative and "simplified" interpretation of finitary forests as forests. So the parameterized proof of decidability rests on three preliminary decidability results, one of which is the decidability of absence of members. Therefore, the latter decidability result stays out of the parameterized scheme and is established separately (but with the same methodology).

The decidability of absence of members also enters in the definition of the *pruned* search space generated from a sequent σ, where all failed runs are chopped off. Pruning is useful because, in our idealized proof search procedure, a run develops fully and in parallel all of its branches, and so a failed run may well be an infinite object. Our final result, which we dub König's lemma for simple types, says that finiteness of all members of a sequent is equivalent to finiteness of the pruned search space generated from the sequent.

References

[1] José Espírito Santo, Ralph Matthes, and Luís Pinto. A coinductive approach to proof search. In David Baelde and Arnaud Carayol, editors, *Proceedings of FICS 2013*, volume 126 of *EPTCS*, pages 28–43, 2013. http://dx.doi.org/10.4204/EPTCS.126.3.

[2] *idem.* Inhabitation in simply-typed lambda-calculus through a lambda-calculus for proof search. In Ambrus Kaposi, editor, *Book of abstracts for TYPES 2017*. http://types2017.elte.hu/proc.pdf.

[3] *idem.* A coinductive approach to proof search through typed lambda-calculi. http://arxiv.org/abs/1602.04382, July 2016.

[4] *idem.* Inhabitation in simply-typed lambda-calculus through a lambda-calculus for proof search. *Mathematical Structures in Computer Science*, pages 1–33, April 2018. First View - volume not yet known, dx.doi.org/10.1017/S0960129518000099.

[5] *idem.* Decidability of several concepts of finiteness for simple types. *Fundamenta Informaticae*, 2019. 28 pages. To appear. Author version at https://hal.archives-ouvertes.fr/hal-02119503.

The Scott Model of PCF in Univalent Type Theory

Tom de Jong

University of Birmingham, United Kingdom

We report on the development of the Scott model of the programming language PCF in constructive predicative univalent type theory. To account for the non-termination in PCF, we work with the partial map classifier monad (also known as the *lifting* monad) from topos theory [6], which has been extended to constructive type theory by Knapp and Escardó [4, 5].

Our results show that lifting is a viable approach to partiality in univalent type theory. Moreover, we show that the Scott model can be constructed in a predicative and constructive setting. Other approaches [1, 3] to partiality either require some form of choice or higher inductive-inductive types. We show that one can do without these extensions.

Capretta's delay monad has been used to give a constructive approach to domain theory [2]. However, the objects have the "wrong equality", so that every object comes with an equivalence relation that maps must preserve. The framework of univalent mathematics in which we have placed our development provides a more natural approach. Moreover, we do not make use of Coq's impredicative `Prop` universe and our treatment incorporates directed complete posets (dcpos) and not just ω-cpos.

Framework. We work in intensional Martin-Löf Type Theory with inductive types (including the empty $\mathbb{0}$, unit $\mathbb{1}$, natural numbers and identity types), $\sum$- and $\prod$-types, functional and propositional extensionality and propositional truncation. We work predicatively, so we do not assume propositional resizing. Although we do not need full univalence at any point, we emphasise the importance of the idea of h-levels, which is fundamental to univalent type theory.

PCF and the Scott model. PCF has a type ι for natural numbers and a function type $\sigma \Rightarrow \tau$ for every two PCF types σ and τ. Every natural number n is represented as the *numeral* $\underline{n}$ of type ι. Moreover, PCF has a fixed point operator. To model this, we work with directed complete posets (dcpos) with a least element.

The lifting $\mathcal{L}(X)$ of a type X is defined as $\sum_{P:\Omega}(P \to X)$, where Ω is a type universe of propositions (subsingletons). Note that we can embed X into $\mathcal{L}(X)$ by $x \mapsto (\mathbb{1}, \lambda t.x)$. If X is a set, then $\mathcal{L}(X)$ is a dcpo and it has a least element given by $(\mathbb{0}, \mathsf{fromempty}_X)$ [5].

We write $[\![\sigma]\!]$ for the interpretation of a PCF type σ, and $[\![t]\!] : [\![\sigma]\!]$ for the interpretation of a PCF term $t : \sigma$. In our model, $[\![\iota]\!] \equiv \mathcal{L}\mathbb{N}$. The function type $\sigma \Rightarrow \tau$ is interpreted as the dcpo of continuous maps from $[\![\sigma]\!]$ to $[\![\tau]\!]$. For a term $s : \sigma \Rightarrow \tau$ and a term $t : \sigma$, the application $(st) : \tau$ is a term, and is interpreted as function application $[\![s]\!]([\![t]\!])$.

The operational semantics of PCF induce a binary *reduction* relation $\triangleright^*$ on terms, where $s \triangleright^* t$ intuitively means that "s computes to t". We show our Scott model to work well with the operational semantics through *soundness* and *computational adequacy*. Soundness means that if $s \triangleright^* t$, then $[\![s]\!] = [\![t]\!]$. Computational adequacy states that for any PCF term t of type ι and natural number n, if $[\![t]\!] = [\![\underline{n}]\!]$, then $t \triangleright^* \underline{n}$.

Computational adequacy and total functionals. An interesting use of computational adequacy is that it allows one to argue semantically to obtain results about termination (i.e. reduction to a numeral) in PCF.

Let σ be a PCF type. A *functional of type* σ is an element of $[\![\sigma]\!]$. By induction on PCF types, we define when a functional is said to be *total*:

(i) a functional i of type ι is total if $i = [\![\underline{n}]\!]$ for some natural number n;

(ii) a functional f of type $\sigma \Rightarrow \tau$ is total if it maps total functionals to total functionals, viz. $f(d)$ is a total functional of type τ for every total functional d of type σ.

Now, let s be a PCF term of type $\sigma_1 \Rightarrow (\sigma_2 \Rightarrow (\cdots(\sigma_n \Rightarrow \iota)\cdots))$. If we can prove that $[\![s]\!]$ is total, then computational adequacy allows us to conclude that for all total inputs $[\![t_1]\!] : [\![\sigma_1]\!], \ldots, [\![t_n]\!] : [\![\sigma_n]\!]$, the term $s(t_1, \ldots, t_n)$ reduces to the numeral representing $[\![s]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$.

Characterising PCF propositions. Recall that PCF terms of type ι are interpreted as elements of the lifting of the natural numbers. Hence, the first projection yields a proposition for every such term. Soundness and computational adequacy allow us to characterise these propositions as those of the form $\exists n : \mathbb{N}(t \rhd^* \underline{n})$, where t is a PCF term of type ι. Intuitively, these propositions are semidecidable, i.e. of the form $\exists n_1, \ldots, n_k : \mathbb{N}(P(n_1, \ldots, n_k))$ where P is a decidable predicate on $\mathbb{N}^k$. In proving this, we are led to study indexed W-types, a particular class of inductive types, and when they have decidable equality. Moreover, we provide some conditions on a relation for its k-step reflexive transitive closure to be decidable.

Formalisation. Most of our results (including soundness and computational adequacy) have been formalised in the proof assistant Coq using the UniMath library [7] and Coq's `Inductive` types. The code may be found at https://github.com/tomdjong/UniMath/tree/paper. The full paper can be found at https://arxiv.org/abs/1904.09810.

Acknowledgements. I would like to thank Martín Escardó for suggesting and supervising this project. I have also benefited from Benedikt Ahrens's support and his help with UniMath.

References

[1] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited: The partiality monad as a quotient inductive-inductive type. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures*, pages 534–549. Springer Berlin Heidelberg, 2017.

[2] Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in coq. In *Lecture Notes in Computer Science*, pages 115–130. Springer Berlin Heidelberg, 2009.

[3] James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the delay monad by weak bisimilarity. *Mathematical Structures in Computer Science*, 29(1):67–92, 2017.

[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] Cory Knapp. *Partial Functions and Recursion in Univalent Type Theory*. PhD thesis, School of Computer Science, University of Birmingham, June 2018.

[6] Anders Kock. Algebras for the partial map classifier monad. In *Lecture Notes in Mathematics*, pages 262–278. Springer Berlin Heidelberg, 1991.

[7] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. https://github.com/UniMath/UniMath.

Closed Inductive-Inductive Types are Reducible to Indexed Inductive Types

Ambrus Kaposi[1], András Kovács[1], and Ambroise Lafont[2]

[1] Eötvös Loránd University, Budapest, Hungary
{akaposi, kovacsandras}@inf.elte.hu
[2] IMT Atlantique, Inria, LS2N CNRS, France
ambroise.lafont@inria.fr

Inductive-inductive types [3, 1] (IITs) allow multiple indexed type families in an inductive definition, where a type family may be indexed over a previously declared one. This enables intrinsically typed formalizations of syntaxes of type theories. We show that IITs are reducible to indexed inductive types in a setting with uniqueness of identity proofs and function extensionality.

Signatures and initial algebras for IITs

In [2], a syntax for signatures of quotient inductive-inductive types (QIITs) is given as a domain-specific type theory, where every typing context can be viewed as a listing of type, value and equality constructors. It is also shown that initial algebras for all QIITs can be constructed from terms of the syntax of signatures.

We can get inductive-inductive signatures by taking the equality-free fragment of QIIT signatures. We also restrict signatures so that they are closed (cannot refer to types external to a signature). In this case it also holds that all IITs are constructible from terms of the syntax of II signatures.

However, this syntax of II signatures is given as a QIIT, and thus at this stage we can only show that if we assume the existence of this particular QIIT, then all IITs exist. We strengthen this result by giving a construction of the syntax of II signatures from indexed inductive types, thereby showing that all IITs are in fact constructible from indexed inductive types.

Constructing the syntax of II signatures

This can be viewed as an instance of the initiality problem popularized by Voevodsky: we have an intrinsically typed categorical notion of model for a particular type theory (the theory of II signatures), and we aim to construct the initial model, using inductively defined preterms and well-formedness relations.

However, the initiality proof in our case is simpler than in general, because the theory of II signatures does not contain β-rules, and thus it is possible to construct the initial model using only β-normal preterms, avoiding the use of quotients.

Hence, we first define normal preterms and typing relations on them, using indexed inductive types. Then, we use well-typed preterms to construct a model of the theory of II signatures. Lastly, we show that the constructed model is initial. We explain the last step in more detail.

Initiality of the term model

Initiality means that we have a *recursion principle* for II signatures, and recursors are also unique. This is less convenient in practice than *induction*, but [2] shows unique recursion to be equivalent to induction, and in our case it is easier to consider the former.

To show initiality, we consider an arbitrary model for the theory of II signatures, and exhibit a unique morphism from the previously given term model to it. We do this in the following steps. Each step is preformed by induction on the presyntax.

1. We define a relation between the presyntax and the given model.
2. We show that the relation is right-unique.
3. We show that the relation is preserved by substitution.
4. We show that the relation is left-total on well-typed presyntax.
5. Since now the relation is shown to be functional, we can use it to build a model morphism.
6. We show that the model morphism is unique.

Streicher's previous construction [4] used a family of partial functions from the presyntax to a given model, which functions are later shown to be total on well-formed presyntax. In contrast, we use a *relation* which is later shown to be functional. For our use case, the relational approach seemed to be more convenient in a mechanized setting, and it could be worthwhile to try it in initiality proofs for other type theories as well.

Formalization

We formalized in Agda (`https://github.com/amblafont/UniversalII/blob/cwf-syntax/Cwf/`) the construction of the syntax of II signatures. Separately, there is an Agda formalization for [2] in `https://bitbucket.org/akaposi/finitaryqiit`, which includes the construction of IITs from the syntax of II signatures.

Note that the [2] formalization assumed definitional computation rules for induction on signatures, while our current construction of signatures only provides propositional computation rules. However, we already assume uniqueness of identity proofs (UIP) along with function extensionality, and also use limited equality reflection in the form of Agda rewrite rules. Hence, our formalization is largely in an extensional setting, and thus the propositional-definitional mismatch is not essential. Repeating these constructions without UIP and equality reflection is a potential line of future work. Another future research would be to extend the current result to open and infinitary signatures and QIITs.

References

[1] Gabe Dijkstra. *Quotient inductive-inductive definitions.* PhD thesis, University of Nottingham, 2017.

[2] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages*, 3(POPL):2, 2019.

[3] Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University, 2013.

[4] Thomas Streicher. *Semantics of type theory: correctness, completeness and independence results.* Springer Science & Business Media, 2012.

A formal, classical proof of the Hahn-Banach theorem

Marie Kerjean[1] and Assia Mahboubi[1]

Inria, LS2N
`Marie.Kerjean@inria.fr, Assia.Mahboubi@inria.fr`

The state of the art in formalized *classical* analysis is probably the corpora available for the HOL-Light and Isabelle/HOL proof assistants [4]. This topic has been much less investigated using proof assistants based on dependent type theory like Coq or Agda. By contrast, the latter make possible the investigation of formalized constructive analysis [8]. In this abstract, we present and discuss a formalization of the Hahn-Banach theorem [9, 2], developed using the Coq proof assistant and based on the Mathematical Components libraries [1], extended with some axioms. The Hahn-Banach Theorem is a cornerstone of functional and convex analysis [10, 7]. Here is the so-called analytical form of the theorem:

Theorem 1. ***Hahn-Banach.*** *Consider V a real vector space, F a sub-vector space of V. Consider p a convex scalar function on V and f a scalar linear map on F. There is a linear scalar map g defined on V, majored by p on V and extending f.*

The convex map p is usually instantiated as a semi-norm in locally convex topological spaces, or as a norm in normed spaces, so as to allow the extension of continuous linear scalar maps. The so-called geometrical form of the Hahn-Banach theorem is a corollary of the latter analytical form. It establishes, for any affine subspace L of a given topological vector space, the existence of a separating affine hyperplane containing L and disjoint from C, an arbitrary non-empty open convex set.

The proof of Theorem 1 goes as follows. By elementary arithmetical computations, and taking benefit of the convexity of p, one shows that a linear partial scalar function bounded by p can always be extended to a real-line not included in its domain. When the co-dimension of F is finite, a recurrence on the co-dimension concludes the proof. When this co-dimension is not known, the existence of a maximal extension is established using a form of choice axiom. For instance, Rudin [10, 3.3.2] considers the collection of pairs (G, g), where G is a vector space containing F and g a scalar linear map defined on G, extending f and bounded by p, and the partial order:

$$(G_1, g_1) \leq (G_2, g_2) \quad \text{iff} \quad G_1 \subseteq G_2 \text{ and } g_1 \leq g_2 \text{ on } G_1.$$

This collection is non-empty, as it contains the pair (F, f), and $\leq$ is a partial order. Rudin then uses Zorn's lemma to conclude.

Up to our best knowledge, there exists few formal proofs of this result in a proof assistant based on some form of type theory. An entry in the Journal of Formalized Mathematics[1] describes its verification using the Mizar proof assistant, which is based on a typed, first-order presentation of set theory. We are also aware of a proof in the Isabelle/HOL proof assistant [3], using the Isar language. Cederquist, Coquand and Negri [6] have given a constructive version of the Hahn-Banach theorem, in a point-free formulation based on formal topology, which seems to have been formalized [5] in the Alf proof assistant, based on Martin-Löf Type Theory (we were not able to retrieve the code and the paper seems unpubished).

In this submission, we propose to discuss the formalization of the analytic form of the Hahn-Banach theorem in the Coq proof assistant, using the Mathematical Components libraries, extended with some axioms. The corresponding Coq file is available at the following location:

[1]`http://mizar.uwb.edu.pl/JFM/Vol5/hahnban.html`

https://github.com/math-comp/analysis/blob/hb/hahn_banach.v

The standard library of the Coq proof assistant includes possible extensions which make classical principles, like choice or excluded middle, available to the user. But the resulting, dependently typed, formal language remains quite different in essence from the one used in Isabelle/HOL, HOL-Light or Mizar. The main point of this submission is to discuss the appropriate formalization choices pertaining to this classical, dependently typed setting.

In this formalization work, we make use of the formal definition of vector spaces available in the Mathematical Components libraries, and we consider a vector space V on a real field $\mathbb{R}$ with a supremum operator. The main issue in this proof is to address the gradual extension of the initial linear application and the possible pitfalls related to partiality. For instance, the explicit handling of sub-vector spaces as done in the aforementioned proof by Rudin [10] can prove quite laborious.

Instead, we consider linear applications defined on the entire space V, and extend gradually the locus where the application is bounded by p. Moreover, instead of considering an order relation on subsets of V, we reason on subsets of $V \times \mathbb{R}$, namely on type $V \to \mathbb{R} \to \text{Prop}$. This type is used to represent the graph of the applications successively considered. We define an appropriate partial order on this type, so as to carry successfully the construction of the successive extensions.

This talk will include a discussion on the axioms added to CIC for the purpose of the proof, namely functional extentionality, propositional extentionality, propositional irrelevance and a choice axiom in the sort `Prop`. We will also discuss the relation with the constructive approach in localic spaces [6].

References

[1] *Mathematical Components.* 2018. https://math-comp.github.io/mcb/.

[2] Stefan Banach. Sur les fonctionnelles linéaires II. *Studia Mathematica*, 1:223–239, 1929.

[3] Gertrud Bauer and Markus Wenzel. Computer-assisted mathematics at work (the Hahn-Banach Theorem in Isabelle/Isar). In *Types for Proofs and Programs, International Workshop TYPES'99, Lökeberg, Sweden, June 12-16, 1999, Selected Papers*, pages 61–76, 1999.

[4] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of real analysis: A survey of proof assistants and libraries. *Mathematical Structures in Computer Science*, 26(7), 2016.

[5] Jan G. Cederquist. A machine-assisted proof of the Hahn-Banach theorem. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.6159&rep=rep1&type=pdf, 1994.

[6] Jan G. Cederquist, Thierry Coquand, and Sara Negri. *The Hahn-Banach Theorem in Type Theory*, pages 57–72. Oxford Univ. Press, 10 1998. Imported from DIES.

[7] Frank H. Clarke. *Optimization and nonsmooth analysis.* Canadian Mathematical Society Series of Monographs and Advanced Texts. John Wiley & Sons, Inc., New York, 1983. A Wiley-Interscience Publication.

[8] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-Corn, the Constructive Coq Repository at Nijmegen. In *Mathematical Knowledge Management, Third International Conference, MKM 2004, Bialowieza, Poland, September 19-21, 2004, Proceedings*, pages 88–103, 2004.

[9] Hans Hahn. Über lineare Gleichungssysteme in linearen Räumen. *Journal für die reine und angewandte Mathematik*, 157:214–229, 1927.

[10] Walter Rudin. *Functional Analysis.* McGraw-Hill Book Company, 1991.

Extracting Exact Bounds from Typing in a Classical Framework

Delia Kesner[1] and Pierre Vial[2]

[1] Univ. de Paris (IRIF, CNRS)
[2] INRIA (LS2N and IMT)

Type systems are designed to capture different properties of programming languages, such as for example termination, data races freedom, memory safety, deadlock freedom, non-interference, privacy, productivity. In this work we use a typing system to provide exact bounds for consumption of resources [5, 2, 1], notably, time (evaluation lengths) and space (size of normal forms). We focus on functional languages with control operators (*e.g.* the `callcc` function), which allow to explicitly alter the control flow of programs. Indeed, control operators can be intuitively seen as `goto` instructions for functional programming, allowing not only bypassing some pieces of code, but also backtracking. It is then not clear how to measure consumption of resources, and in particular evaluation lengths. We overcome the challenge of this goal by making use of non-idempotent types.

Indeed, our approach is based on *intersection and union types* [3, 4, 7], which are able to fully characterize normalization properties, in the sense that a term is typable *if and only if* it is normalizing. More precisely, while *idempotent* intersection and union types provide only *qualitative* characterizations as the one mentioned above, we use here *non-idempotent* ones [6, 5], which have the power to provide *quantitative* characterizations, in the sense that typable terms provide also upper or exact bounds for normalization evaluation lengths and/or normal form sizes. Giving *exact bounds* for computations by means of typing is challenging, particularly because the typing system needs to statically understand the main ingredients to measure evaluation, which is a dynamic property. Notably, our typing system gives at the same time exact measures for the number of evaluation steps *and* the size of the result, and these integers are measured *separately*, simply because the size of the result can be exponentially bigger than the number of steps.

More specifically, we capture the power of functional languages with control operators by means of the lambda-mu calculus of M. Parigot [8], for which exact bounds, and even upper bounds, are not easy to establish. From a technical point of view our typing judgements are decorated with three counters, one to measure β-evaluation lengths, the second to measure μ-evaluation lengths and a last one to measure the size of the result of the computation. Our typing system makes use of several key tools, the most important of which are the following:

- **Persistent and consuming arrows**. We distinguish between two different kind of functional arrows: $\mathcal{I} \rightarrow \mathcal{U}$ is a functional type made by a *consuming arrow*, which types an application node that will be consumed during evaluation, while $\mathcal{I} \twoheadrightarrow \mathcal{U}$ is a functional type made by a *persistent arrow*, which types a persistent application, *i.e.* an application which contributes to the size of the normal form. Building on this idea, normal (resp. not normal) applications terms will be typed with persistent (resp. consuming) arrows. Consequently, only persistent arrows contribute to the size of normal forms, and only consuming arrows contribute to evaluation lengths.

- **Activation operator**. The type of a μ-abstraction $\mu\alpha.c$ carries the type of all the commands named α inside c, and particularly, those of *neutral* normal forms (normal forms which are not abstractions). Consuming arguments for such a μ-abstraction creates new application nodes but without creating any new redex. Intuitively, the type of such neutral terms should be persistent (contributing to the normal form) whereas the type of $\mu\alpha.c$ should be consuming (contributing to μ-evaluation). This mismatch explains the introduction of an *activation operator*, which transforms persistent functions types (pertaining to neutral terms in commands) into consuming ones (carried by $\mu\alpha.c$).

The main contributions of this work are the correctness and completeness results for typing of functional programming with control operators w.r.t. evaluation. More precisely,

- *Correctness*: every (tight) typable term with counters (ℓ, m, f) evaluates to a normal form of size f by means of ℓ β-steps and m μ-steps.
- *Completeness*: every term evaluating to a normal form of size f by means of ℓ β-steps and m μ-steps is a (tight) typable term with counters (ℓ, m, f).

Our development is carried out for three different evaluation strategies: head-evaluation, leftmost-outermost evaluation and maximal evaluation, each of them characterizing, respectively, head, weak and strong normalization in the $\lambda\mu$-calculus.

However, we do not provide one typing system per strategy, but only a *unique parametrized* system, that can be instantiated for different cases, namely, head, leftmost and maximal evaluation. This allows us to *factorize* the proofs of the properties pertaining to the three mentioned strategies. This new approach emphasizes the differences between the different systems for the mentioned reduction strategies, by putting together at the same time all their resemblances.

References

[1] B. Accattoli, S. Graham-Lengrand, and D. Kesner. Tight typings and split bounds. *PACMPL*, 2(ICFP):94:1–94:30, 2018.

[2] A. Bernadet and S. Lengrand. Complexity of strongly normalising λ-terms via non-idempotent intersection types. In M. Hofmann, editor, *F*oundations of Software Science and Computation Structures (FOSSACS), volume 6604 of *L*ecture Notes in Computer Science. Springer-Verlag, 2011.

[3] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for lambda-terms. *A*rchive for Mathematical Logic, 19:139–156, 1978.

[4] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *N*otre Dame Journal of Formal Logic, 4:685–693, 1980.

[5] D. de Carvalho. *S*émantique de la logique linéaire et temps de calcul. PhD thesis, Université Aix-Marseille, Nov. 2007.

[6] P. Gardner. Discovering needed reductions using type theory. In *TACS*, Sendai, 1994.

[7] O. Laurent. On the denotational semantics of the untyped lambda-mu calculus, 2004. Unpublished note.

[8] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *LPAR*, pages 190–201, 1992.

Dependent Event Types in Event Semantics*

Zhaohui Luo[1] and Sergei Soloviev[2]

[1] Royal Holloway, University of London, U.K.
zhaohui.luo@hotmail.co.uk
[2] IRIT, Toulouse, France
Sergei.Soloviev@irit.fr

It was pointed out recently [1, 7] that refining event types by dependent types and subtyping is quite useful in analysis of some difficult cases in Davidson's event semantics [4, 9]. In particular, this has led to a satisfactory solution to the event quantification problem [2, 10, 5], concerning incompatibilities that arise in combining event semantics with the traditional Montague semantics, about scopes when existential event quantifiers interact with other quantifiers.

The sentence (1) may be seen as a typical example, which contains the general quantifier 'no'. Consider its possible interpretations (2) and (3), where *Event* is the type of all events, **e** type of all entities, $dog : \mathbf{e} \to \mathbf{t}$ a predicate from entities to truth values (of type **t**), $bark : Event \to \mathbf{t}$ a predicate over events, and $agent : Event \to \mathbf{e} \to \mathbf{t}$ a thematic relation between events and entities with $agent(e, x)$ meaning that x acts in event e.[1]

(1) No dog barked.

(2) $\neg\exists x : \mathbf{e}.\ (dog(x)\ \&\ \exists v{:}Event.\ (bark(v)\ \&\ agent(v, x)))$

(3) $\exists v : Event.\ \neg\exists x : \mathbf{e}.\ (dog(x)\ \&\ bark(v)\ \&\ agent(v, x))$

Note that the second interpretation (3) is incorrect from the commonsense viewpoint: it means that 'either there was a non-barking event or there was a barking event whose agent was not a dog'. However, formally, the incorrect interpretation (3) is acceptable just as the correct one (2): (3) is a legal formula.

In the standard Montagovian setting, the Davidson's event semantics has only one type *Event* of all events, but the above event quantification problem cannot be resolved merely by introduction of many non-dependent types of events. Rather, as shown in [7], dependent types combined with subtyping offer a natural solution.

Traditional Davidsonian semantics is built on the basis of Church's simple type theory as employed in Montague semantics. We consider two extensions with dependent event types (depending on thematic roles, such as *agent* and *patient*): first, the extension $\mathcal{C}_e$ of Church's simple type theory by appropriate dependent types and predicates and, secondly, the extension of a modern type theory (MTT) as employed in MTT-semantics [6, 3]. As in [8], we denote this extension $T[E]$, where T is an MTT and E the set of basic coercions describing subtyping relations between dependent event types.

Rather than a single type *Event* of events, we introduce types of events that are dependent on some parameters. For instance, an event type can be dependent on agents and patients. Let *Agent* and *Patient* be the types of agents and patients, respectively. Then, for $a : Agent$ and $p : Patient$, the dependent type $Evt_{AP}(a, p)$ is the type of events whose agents are a and whose patients are p, and $Evt_A(a)$ is that of events whose agents are a.[2]

*Partially supported by EU COST Action CA15123.

[1]Other thematic relations may be considered; e.g., $patient(e, x)$ means that x is a target of an action in e.

[2]Technically, in our framework, the thematic relations such as *agent* and *patient* can be defined. In other words, one does not need to assume such relations anymore.

The relationships between dependent event types are characterised by subtyping (subsumptive for $\mathcal{C}_e$ and coercive for $T[E]$), which provide consistency for interpretations that may use either a supertype or a subtype. Consider again the sentence (1). In MTT-semantics common nouns are interpreted as types (rather than predicates). One may translate' the interpretations (2) and (3) into (4) and (5), respectively (the relation *agent* disappears: see Fn 2):

(4) $\neg\exists x : Dog.\ \exists v : Evt_A(x).\ bark(v)$.

(5) $\exists v : Evt_A(x).\ \neg\exists x : Dog.\ bark(v)$

Here, 'literally', $bark(v)$ would be ill-typed since $Evt_A(x)$ is not $Event$, while $bark : Event \rightarrow \mathbf{t}$. However, with subtyping, $bark(v)$ is well-typed since $bark : Event \rightarrow \mathbf{t} \leq Evet_A(x) \rightarrow \mathbf{t}$, for $x : Dog \leq Agent$.

Here (5) is not just incorrect, but ill-typed, because the first x is a variable that is not assumed anywhere, while the correct interpretation is well-typed.

The formalisation of this example in $T[E]$ is similar, and the incorrect interpretation will also be ill-typed for the same reason, the only difference being that we adopt coercive subtyping rather than subsumptive subtyping. A standard requirement [8] is coherence of the set E of basic coercions: e.g., if we consider the chains of coercions $Evt_{AP}(a,p) <_{c_1} Evt_A(a) <_{c_2} Event$ and $Evt_{AP}(a,p) <_{c'_1} Evt_P(p) <_{c'_2} Event$ then $c_2 \circ c_1 = c'_2 \circ c'_1$.

With respect to [7] the talk will present the proofs of the following two new results.

Theorem 1. The system $\mathcal{C}_e$ with dependent event types and subsumptive subtyping is a conservative extension of Church's simple type theory.

Theorem 2. Let T be a type theory specified in LF and E the set of coercions between dependent event types. Then, E is coherent and, hence, the type theory $T[E]$, the extension of T by coercive subtyping specified in E, as described in [8], is a conservative extension of T.

References

[1] N. Asher and Z. Luo. Formalisation of coercions in lexical semantics. *Sinn und Bedeutung 17, Paris*, 2012.

[2] L. Champollion. The interaction of compositional semantics and event semantics. *Linguistics and Philosophy*, 38:31–66, 2015.

[3] S. Chatzikyriakidis and Z. Luo. *Formal Semantics in Modern Type Theories.* Wiley & ISTE Science Publishing Ltd., 2019. (to appear).

[4] D. Davidson. The logical form of action sentences. *In: S. Rothstein (ed.). The Logic of Decision and Action. University of Pittsburgh Press*, 1967.

[5] P. de Groote and Y. Winter. A type-logical account of quantification in event semantics. *Logic and Engineering of Natural Language Semantics 11*, 2014.

[6] Z. Luo. Formal semantics in modern type theories with coercive subtyping. *Linguistics and Philosophy*, 35(6):491–513, 2012.

[7] Z. Luo and S. Soloviev. Dependent event types. In *WoLLIC 2017, LNCS 10388*, pages 274–287.

[8] Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: theory and implementation. *Information and Computation*, 223:18–42, 2012.

[9] T. Parsons. *Events in the Semantics of English.* MIT Press, 1990.

[10] Y. Winter and J. Zwarts. Event semantics and abstract categorial grammar. *Proc. of Mathematics of Language 12, LNCS 6878*, 2011.

Universal Algebra in HoTT

Andreas Lynge[1] and Bas Spitters[2]

[1] Aarhus University, Aarhus, Denmark
andreaslynge@cs.au.dk
[2] Aarhus University, Aarhus, Denmark
b.a.w.spitters@gmail.com

Introduction

Universal algebra is a mathematical theory of algebraic structures. The isomorphism theorems in universal algebra are generalizations of the isomorphism theorems known from group theory and ring theory. In universal algebra these theorems apply to all algebras, e.g. groups, rings, groups acting on sets, etc.

Universal algebra has been developed in type theory before [6, 4, 2]. To model quotient types and function extensionality, these developments are using setoids. This leads to well-known problems.

We formalize universal algebra in the HoTT library [1] using Coq's type class mechanism [5] in the style of the math-classes library [6]. Propositional truncation and quotient types are defined in terms of HITs [7, Chapter 6] and the univalence axiom implies function extensionality [7, Section 4.9]. By using this we avoid the need for setoids. We show that there is a univalent category of algebras and homomorphisms for a signature. The development contains the three fundamental isomorphism theorems, which become identification theorems in HoTT.

The following sections give a brief overview of the work. A longer explanation is available at https://github.com/andreaslyn/Work/blob/master/Math-Bachelor.pdf.

Fundamental definitions

A (multi-sorted) *algebra* A : Algebra(σ) for a signature σ : Signature consists of

- A *carrier* type $A_s : \mathcal{U}$ for each $s : \mathrm{Sort}(\sigma)$, where $\mathrm{Sort}(\sigma)$ is a type of sorts corresponding to the signature σ. It is required that A_s is a set for all $s : \mathrm{Sort}(\sigma)$.
- An *operation* $u^A : \mathrm{Operation}(A, u)$ for each $u : \mathrm{Symbol}(\sigma)$. Here $\mathrm{Symbol}(\sigma)$ is a type of function symbols corresponding to σ, and

$$\mathrm{Operation}(A, u) \equiv (A_{s_1} \to A_{s_2} \to \cdots \to A_{s_n}),$$

  where $n : \mathbb{N}$ and $s_1, \ldots, s_n : \mathrm{Sort}(\sigma)$ depends on u.

For example, a group G acting on a set S is an algebra with two carrier types, the group G and the set S. This algebra has the usual group operations: the identity element $e : G$, the binary operation $\cdot : G \to G \to G$, and the inverse operation $(-)^{-1} : G \to G$. Additionally there is the action of G on S, an operation $\alpha : G \to S \to S$.

A *homomorphism* $f : A \to B$ between algebras $A, B : \mathrm{Algebra}(\sigma)$ is a family of functions $f_s : A_s \to B_s$, indexed by $s : \mathrm{Sort}(\sigma)$, that preserves operations in the sense that

$$f_{s_{n+1}}(u^A(x_1, x_2, \ldots, x_n)) = u^B(f_{s_1}(x_1), f_{s_2}(x_2), \ldots, f_{s_n}(x_n)),$$

for all $u : \mathrm{Symbol}(\sigma)$ and $x_i : A_{s_i}$.

An *isomorphism* is a homomorphism $f : A \to B$ where, for all $s : \mathrm{Sort}(\sigma)$, f_s is both injective and surjective, or equivalently f_s is an equivalence.

A property of homomorphisms $f : A \to B$ is that equational laws involving operations, such as $u^A(v^A(x), y) = w^A(x, y)$, are always preserved,

$$u^B(v^B(f_r(x)), f_s(y)) = f_t(u^A(v^A(x), y)) = f_t(w^A(x, y)) = w^B(f_r(x), f_s(y)).$$

Results

For generic single-sorted (single carrier type) algebraic structures, Coquand and Danielsson show that isomorphic structures are equal [3]. This leads us to a central theorem about multi-sorted algebras:

If there is an isomorphism $A \to B$ between two algebras $A, B : \mathrm{Algebra}(\sigma)$, then $A = B$.

This is in fact an equivalence, which we use to show that there is a (univalent) category σ-**Alg** of algebras and homomorphisms for signature σ. This was previously formalized in HoTT for single-sorted algebraic structures [7, Section 9.8]. We have generalized this to multi-sorted algebraic structures, but with a more specific notion of algebraic structure and homomorphism.

We define product algebra, subalgebra and quotient algebra. Product algebras are used to construct products in the category σ-**Alg**, equalisers are subalgebras, and coequalisers are quotient algebras. We prove the three isomorphism theorems. The first isomorphism theorem states that:

Given a homomorphism $f : A \to B$ between algebras $A, B : \mathrm{Algebra}(\sigma)$,

- *The kernel of f, defined by $\ker(f)(s, x, y) :\equiv (f_s(x) = f_s(y))$, gives rise to a quotient algebra $A/\ker(f)$ of A by $\ker(f)$.*
- *Set $\mathrm{inim}(f)(s, y) :\equiv \big\| \sum_{(x:A_s)}(f_s(x) = y)\big\|$, where $\|-\|$ denotes propositional truncation. It induces a subalgebra $B\&\,\mathrm{inim}(f)$ of B, the homomorphic image of f.*
- *There is an isomorphism $A/\ker(f) \to B\&\,\mathrm{inim}(f)$, and hence $A/\ker(f) = B\&\,\mathrm{inim}(f)$.*

It follows that any morphism $f : A \to B$ in σ-**Alg** image factorizes $A \to B\&\,\mathrm{inim}(f) \hookrightarrow B$. Images are stable under pullback, so the category σ-**Alg** of algebras for signature σ is regular.

References

[1] A. Bauer, J. Gross, P. L. Lumsdaine, M. Shulman, M. Sozeau, and B. Spitters. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 164–172. ACM, 2017. http://doi.acm.org/10.1145/3018610.3018615.

[2] V. Capretta. Universal Algebra in Type Theory. In Y. Bertot, G. Dowek, A. Hirschowits, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *LNCS*, pages 131–148. Springer, 1999.

[3] T. Coquand and N. A. Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105 – 1120, 2013. In memory of N.G. (Dick) de Bruijn (19182012), http://www.sciencedirect.com/science/article/pii/S0019357713000694.

[4] E. Gunther, A. Gadea, and M. Pagano. Formalization of Universal Algebra in Agda. *Electronic Notes in Theoretical Computer Science*, 338:147–166, 10 2018.

[5] M. Sozeau and N. Oury. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 278–293, Berlin, 2008. Springer. http://dx.doi.org/10.1007/978-3-540-71067-7_23.

[6] B. Spitters and E. van der Weegen. Type Classes for Mathematics in Type Theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011.

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

Dijkstra Monads for All
(Extended Abstract)

Kenji Maillard[1,2], Danel Ahman[3], Robert Atkey[4], Guido Martínez[5], Cătălin Hriţcu[1], Exequiel Rivas[1], and Éric Tanter[1,6]

[1]Inria Paris, [2]ENS Paris, [3]University of Ljubljana, [4]University of Strathclyde, [5]CIFASIS-CONICET Rosario, [6]University of Chile

We propose a semantic framework [4] for specifying and verifying programs with side-effects modeled by computational monads [5]. The framework is based on Dijkstra monads, which have proven valuable in practice for verifying effectful code [7]. A *Dijkstra monad* $\mathcal{D}\ A\ w$ is a monad-like structure that classifies programs returning values in A and specified by $w : WA$, where W is what we call a *specification monad*. A typical specification monad contains predicate transformers mapping postconditions to preconditions, and is naturally ordered by pointwise implication. For instance, for code in the state monad $\mathrm{St}\,A = S \to A \times S$, a natural specification monad is $W^{\mathrm{St}}A = (A \times S \to \mathbb{P}) \to (S \to \mathbb{P})$, mapping postconditions (on return values and final states) to preconditions (on the initial states).

Our work [4] was driven by the following two questions: How to associate a specification monad W to an arbitrary computational monad M? And once we have a computational monad M and a corresponding specification monad W, can we construct a Dijkstra monad out of them?

A partial answer to these questions was given in *Dijkstra Monads for Free* (*DM4Free*) [1]: from a computational monad defined as a term in a metalanguage called DM, a canonical specification monad is automatically derived through a syntactic translation. We observe that a monad in DM essentially yields a monad transformer $\mathcal{T}$, giving rise to both a computational monad $\mathcal{T}(\mathrm{Id})$, by applying it to the identity monad, and a specification monad $\mathcal{T}(W^{\mathrm{Pure}})$, by applying it to the continuation monad $W^{\mathrm{Pure}}A = (A \to \mathbb{P}) \to \mathbb{P}$. Returning to our example, $W^{\mathrm{St}}A$ can be obtained by applying the monad transformer $\mathrm{StT}\,M\,A = S \to M(A \times S)$ to W^{Pure}. Consequently, the class of supported computational monads is restricted to those that can be decomposed as a monad transformer applied to Id, ruling out various effects such as nondeterminism or IO, for which no proper monad transformer is known [3].

To overcome this limitation, we decouple the computational monad and the specification monad: instead of insisting on deriving both monads from the same monad transformer as in *DM4Free*, we consider them independently and only require that they are suitably related, namely, by a *monad morphism*.

We illustrate on the computational monad for interactive IO that neither the specification monad nor the monad morphisms need to be canonical. The IO monad is represented as trees of interactions $\mathrm{IO}\,A = \mu Y. A + (I \to Y) + O \times Y$. It is the free monad on two operations: $\mathtt{read} : \mathrm{IO}\,I$, that reads an input of type I, and $\mathtt{write} : O \to \mathrm{IO}\,\mathbb{1}$, that outputs an element of type O. As IO is a free monad, a monad morphism out of it is defined by its action on operations. Considering the simple specification monad W^{Pure}, we can define two monad morphisms $\theta^{\forall}, \theta^{\exists} : \mathrm{IO} \to W^{\mathrm{Pure}}$, differing on read:

$$\theta^{\forall}(\mathtt{read}) = \lambda(p : I \to \mathbb{P}).\, \forall i. p\, i \ :\ W^{\mathrm{Pure}}\, I \qquad \theta^{\exists}(\mathtt{read}) = \lambda(p : I \to \mathbb{P}).\, \exists i. p\, i \ :\ W^{\mathrm{Pure}}\, I$$

whereas $\theta^{\forall/\exists}(\mathtt{write}\, o)$ forgets the write via $\lambda(p : \mathbb{1} \to \mathbb{P}).\ p\, * \ :\ W^{\mathrm{Pure}}\, \mathbb{1}$. $\theta^{\forall}$ mandates that a program should satisfy the postcondition for **all** input values, whereas $\theta^{\exists}$ accepts programs that satisfy the postcondition for **some** input value, analogously to the two modalities of evaluation logic [6].

To provide specifications aware of the output, we apply an update monad transformer [2] to W^{Pure}, obtaining the specification monad $W^{\mathrm{Hist}}\, X = (X \times \mathcal{E}^* \to \mathbb{P}) \to \mathcal{E}^* \to \mathbb{P}$ where $\mathcal{E} = \mathrm{In}\, I \mid \mathrm{Out}\, O$ is the alphabet of IO events. Specifications can now refer to both the history of past events and the trace of events generated by the program being specified. A monad morphism θ^{Hist} from IO is given by:

$$\theta^{\mathrm{Hist}}(\mathtt{read}) = \lambda p\, h.\, \forall i, p\, \langle i, [\mathrm{In}\, i]\rangle \ :\ W^{\mathrm{Hist}}(I) \qquad \theta^{\mathrm{Hist}}(\mathtt{write}\, o) = \lambda p\, h.\, p\, \langle *, [\mathrm{Out}\, o]\rangle \ :\ W^{\mathrm{Hist}}(\mathbb{1})$$

While W^{Hist} provides a good way to reason about IO, some IO programs do not depend on the context of past interactions. We can provide an even more parsimonious way to specify and verify such programs by applying a writer transformer to W^{Pure}. The resulting specification monad W^{Fr} is in fact a special case of W^{Hist} when the history is taken to be $\mathbb{1}$ [2]. Analogously to the above, a monad morphism θ^{Fr} similar to θ^{Hist} can be defined. These four possibilities of specifying IO can be summarised as follows:

$$\begin{tikzcd}
 & & W^{\mathrm{Hist}}\,X = (X \times \mathcal{E}^* \to \mathbb{P}) \to \mathcal{E}^* \to \mathbb{P} \\
W^{\mathrm{Pure}} & \mathrm{IO} \arrow[l, bend right, "\theta^{\forall}"'] \arrow[l, bend left, "\theta^{\exists}"] \arrow[ur, bend left, "\theta^{\mathrm{Hist}}"] \arrow[dr, bend right, "\theta^{\mathrm{Fr}}"'] & \\
 & & W^{\mathrm{Fr}}\,X = (X \times \mathcal{E}^* \to \mathbb{P}) \to \mathbb{P}
\end{tikzcd}$$

With similar ease, we can also use computational and specification monads to reason about combinations of effects. For example, to reason about computations that can both perform IO and manipulate state, we can pair the computational monad $\mathrm{IOSt}\,X = S \to \mathrm{IO}(X \times S)$ with the specification monad $W^{\mathrm{IOSt}}\,X = (X \times S \times \mathcal{E}^* \to \mathbb{P}) \to S \to \mathcal{E}^* \to \mathbb{P}$, with the corresponding monad morphism θ^{IOSt} naturally combining the behaviour of θ^{Hist} and θ^{St}, where $\theta^{\mathrm{St}}(f) = \lambda p\, s.\, p\,(f\,s) : \mathrm{St} \to W^{\mathrm{St}}$. Further, there exist various natural pairings (via monad morphisms) of computational monads and specification monads for nondeterminism, exceptions, partiality and their combinations.

A key contribution of our work is the observation that Dijkstra monads can be reconstructed from such monad morphisms: given a monad morphism $\theta : M \to W$, a type A, and a specification $w : W\,A$, we can consider the subtype $\mathcal{D}^M A\,w = \{m : M\,A \mid w \leq^W \theta(m)\}$ of computations in M returning values in A and specified by w. This assignment turns out to always provide a Dijkstra monad, and is in fact part of a (categorical) equivalence between categories of Dijkstra monads and monad morphisms.

We illustrate the applicability of this correspondence on θ^{Fr} and θ^{IOSt}: we define two Dijkstra monads, $\mathrm{IOFree}\;A\;(w : W^{\mathrm{Fr}}\,A)$ and $\mathrm{IOST}\;A\;(w : W^{\mathrm{IOSt}}\,A)$, and consider the following programs:

```
let duplicate () : IOFree unit (λ p → ∀i. p ((), [In i; Out i; Out i])) =
  let i = read () in write i; write i

let do_io_then_roll_back_state () : IOST unit (λ p s h → ∀i . p (() , s , [In i; Out (s + i + 1)])) =
  let s = get () in let i = read () in put (s + i); let s' = get () in write (s' + 1); put s
```

The first program reads the input once and then outputs the read value twice, exactly as expressed in its specification. The second program combines IO and state, mutating the state to compute the output from the input, but then rolls back the state to its initial value, again as described by its specification.

The loose coupling between computational and specification monads described here and the corresponding draft paper [4], via monad morphisms, provides great flexibility in choosing the most suitable candidates for the verification task at hand. We have implemented this framework both in Coq[1] and F*[2], and verified example programs using the Dijkstra monads induced by such monad morphisms.

References

[1] D. Ahman et al. "Dijkstra Monads for Free". In: *POPL 17*. ACM, Jan. 2017.

[2] D. Ahman et al. "Update Monads: Cointerpreting Directed Containers". In: *TYPES 13*.

[3] M. Hyland et al. "Combining algebraic effects with continuations". In: *T.C.S.* (2007).

[4] K. Maillard et al. *Dijkstra Monads for All*. Draft at https://arxiv.org/abs/1903.01237. Mar. 2019.

[5] E. Moggi. "Computational Lambda-Calculus and Monads". In: *LICS '89*. 1989.

[6] A. M Pitts. "Evaluation logic". In: *IV Higher Order Workshop*. Springer. 1991.

[7] N. Swamy et al. "Dependent Types and Multi-Monadic Effects in F*". In: *POPL '16*.

[1] https://gitlab.inria.fr/kmaillar/dijkstra-monads-for-all

[2] https://github.com/FStarLang/FStar/tree/guido_effects/examples/dm4all

How Erasure is Linked to (im)Predicativity

Stefan Monnier and Nathaniel Bos

Université de Montréal - DIRO
monnier@iro.umontreal.ca   nathaniel.bos@mail.mcgill.ca

Abstract

Of all the threats to the consistency of a type system, such as side effects and recursion, impredicativity is arguably the least understood. In this paper, we revisit several type systems which do include some form of impredicativity and show that they can be refined to equivalent systems where impredicative quantification can be erased, in a stricter sense than the kind of proof irrelevance notion used for example for Prop terms in systems like Coq.

We hope these observations will lead to a better understanding of why and when impredicativity can be sound. As a first step in this direction, we discuss how these results suggest some extensions of those systems which might still enjoy consistency.

1 Introduction

Diagonalization proofs and paradoxes such as "This sentence is false" show the dangers of self reference: admitting such propositions in a logic leads to inconsistencies. For this reason Russell introduced the concept of *type* as well as *predicativity* (and its inverse).

The stratification enforced by predicativity seems sufficient to protect us from such paradoxes, but it does not seem to be absolutely necessary either: systems such as System-F are not predicative yet they are generally believed to be consistent. Some people reject impredicativity outright, and indeed systems like Agda [Bove et al., 2009] demonstrate that impredicativity is not indispensable to get a powerful logic. Yet, many popular systems, like Coq [Huet et al., 2000], do include some limited form of impredicativity, although those limits tend to feel somewhat ad-hoc, making the overall system more complex, with unsatisfying corner cases. For this reason we feel there is a need to try and better understand what those limits to impredicativity should look like.

Let's disappoint the optimistic reader right away: this paper does not solve this problem. But during the design of our experimental language Typer [Monnier, 2019], we noticed an interesting property shared by several existing impredicative systems which are believed to be consistent, that seemed to link impredicativity and erasability. Some mathematicians, such as Carnap [Fruchart and Longo, 1996], have argued that impredicative quantification might be acceptable as long as those arguments are not used in what we could describe as a "significant" way, so we investigate here whether erasability might be such a notion of "insignificance".

Arguably, such a link between impredicativity and erasure is already evidenced by systems like Coq whose impredicative universe is also erasable, and even more so by the propositional resizing axiom [The Univalent Foundations Program, 2013] which allows impredicativity for all *mere propositions*, i.e. types whose inhabitants are all provably equal and hence erasable. But we here link impredicativity to the somewhat different notion of erasability of arguments which are only used in type annotations, such as the *implicit* arguments in ICC* and EPTS [Barras and Bernardo, 2008, Mishra-Linger and Sheard, 2008].

More specifically, we take various impredicative systems and refine them with annotations of *erasability*, and then show that all impredicative quantifications can be annotated as erasable.

Armed with this proverbial hammer, we then look at a few other forms of impredicativity that are known to break consistency and argue that they look like nails: by restricting those forms of impredicativity to be erasable we may be able to recover consistency.

The contributions of this work are:

- A proof that in CCω all impredicative functions actually take erasable arguments.
- A proof that in CIC all impredicative functions take erasable arguments and that all *large* fields of inductive types are also erasable.
- A potentially consistent extension of CIC with strong elimination of large inductive types.
- A proof that the same idea does not allow impredicativity in more than one universe.
- As needed for some of the above contributions, we sketch a calculus with both inductive types and erasability annotations. While this is straightforward, we do not know of such a system published so far, the closest we found being the one described by Bernardo [2009].

References

Bruno Barras and Bruno Bernardo. Implicit calculus of constructions as a programming language with dependent types. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *LNCS*, April 2008.

Bruno Bernardo. Towards an implicit calculus of inductive constructions. extending the implicit calculus of constructions with union and subset types. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 5674 of *LNCS*, August 2009.

Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 5674 of *LNCS*, pages 73–78, August 2009.

Thomas Fruchart and Guiseppe Longo. Carnap's remarks on impredicative definitions and the genericity theorem. Technical Report LIENS-96-22, ENS, Paris, 1996.

Gérard P. Huet, Christine Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.

Alexandre Miquel. The implicit calculus of constructions: extending pure type systems with an intersection type binder and subtyping. In *International conference on Typed Lambda Calculi and Applications*, pages 344–359, 2001.

Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *LNCS*, pages 350–364, April 2008.

Stefan Monnier. Typer: ML boosted with type theory and Scheme. In *Journées Francophones des Langages Applicatifs*, pages 193–208, 2019.

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

A Double-Categorical Perspective on Type Universes

Edward Morehouse, G. A. Kavvos[*] and Daniel R. Licata[*]

Wesleyan University, Middletown, Connecticut, USA

In type theory, the concept of *universe* provides a comprehension structure for types, in the sense that a universe is a type whose inhabitants are themselves types. Semantics for type universes can be given in appropriate sorts of categories.

In higher-dimensional type theory there is more than one sort of map between types that we want to consider. For example, in Homotopy Type Theory (HoTT) we have functions between types, and also equalities between types which, by the Univalence axiom, are equivalent to functions that take part in an equivalence. Thus it would be useful to have a categorical semantics in which we have more than one sort of morphism: one to interpret functions between types and another to interpret paths between types. Moreover, we would like these two sorts of morphism to be related by a Univalence-like principle.

In HoTT, the paths between types are symmetrical, but in directed type theories, e.g. [3], this need not be the case. We are interested in studying categorical models that allow, but do not require, such directional symmetry of paths. However, we wish to retain one of the most basic aspects of HoTT's Univalence for identity types: whenever we have a path between types, we can *coerce* an element of its domain type along it in order to obtain an element of its codomain type. As a slogan, "all paths are coercible".

A *double category* is made up of objects, two distinct sorts of morphisms, which we call "arrows" (with homs "$- \to -$" and units "id") and "proarrows" (with homs "$- \nrightarrow -$" and units "U"), and squares whose opposite faces are morphisms of the same sort and whose adjacent faces are morphisms of opposite sorts. Squares compose in both dimensions by pasting, and any way of pasting together a composable diagram yields the same composite[1].

We can use the arrows and proarrows of a double category to interpret the functions and paths of a simple – i.e. non-dependent – directed type theory whose types are interpreted as categories. The two dimensions are related by the property that all paths arise from functions. This is what makes them coercible: to coerce along a path, simply apply the corresponding function. Although all paths arise from functions, not all functions need give rise to paths. By varying which functions do, we can construct models of universes with different properties.

Any such model satisfies the following Univalence-like principle: the bicategory of types, path-determining functions and function homotopies between them is biequivalent to that of types, paths and path homotopies between them. The structure interpreting this link is that of companions.

In a double category, an arrow $f : \mathrm{A} \to \mathrm{B}$ and proarrow $\mathrm{M} : \mathrm{A} \nrightarrow \mathrm{B}$ are *companions*[2] if there are squares

$$\begin{tikzcd} \mathrm{A} \arrow[r, "\mathrm{M}", "\shortmid"{marking}] \arrow[d, "f"'] \arrow[dr, phantom, "\llcorner f \lrcorner"] & \mathrm{B} \arrow[d, "\mathrm{id}"] \\ \mathrm{B} \arrow[r, "\mathrm{U}"', "\shortmid"{marking}] & \mathrm{B} \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} \mathrm{A} \arrow[r, "\mathrm{U}", "\shortmid"{marking}] \arrow[d, "\mathrm{id}"'] \arrow[dr, phantom, "\ulcorner f \urcorner"] & \mathrm{A} \arrow[d, "f"] \\ \mathrm{A} \arrow[r, "\mathrm{M}"', "\shortmid"{marking}] & \mathrm{B} \end{tikzcd}$$

[*]This material is based on research sponsored by The United States Air Force Research Laboratory under agreement number FA9550-15-1-0053 and FA9550-16-1-0292. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government, or Carnegie Mellon University.

satisfying the following equations (up to natural isomorphism):

$$\begin{tikzcd}
A \arrow[r, "U", "|" marking] \arrow[d, "\mathrm{id}"'] \arrow[dr, phantom, "\ulcorner f."] & A \arrow[d, "f"] \\
A \arrow[r, dashed, "M" description] \arrow[d, "f"'] \arrow[dr, phantom, "\cdot f \lrcorner"] & B \arrow[d, "\mathrm{id}"] \\
B \arrow[r, "U"', "|" marking] & B
\end{tikzcd}
=
\begin{tikzcd}
A \arrow[r, "U", "|" marking] \arrow[d, "f"'] \arrow[dr, phantom, "U"] & A \arrow[d, "f"] \\
B \arrow[r, "U"', "|" marking] & B
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
A \arrow[r, "U", "|" marking] \arrow[d, "\mathrm{id}"'] \arrow[dr, phantom, "\ulcorner f."] & A \arrow[r, "M", "|" marking] \arrow[d, dashed, "f" description] \arrow[dr, phantom, "\cdot f \lrcorner"] & B \arrow[d, "\mathrm{id}"] \\
A \arrow[r, "M"', "|" marking] & B \arrow[r, "U"', "|" marking] & B
\end{tikzcd}
=
\begin{tikzcd}
A \arrow[r, "M", "|" marking] \arrow[d, "\mathrm{id}"'] \arrow[dr, phantom, "\mathrm{id}"] & B \arrow[d, "\mathrm{id}"] \\
A \arrow[r, "M"', "|" marking] & B
\end{tikzcd}$$

Companions, when they exist, are unique up to a canonical isomorphism. We write "$\hat{f}$" for the companion proarrow (interpreting a path) of an arrow f (interpreting a function). The type-theoretic property that all paths are coercible corresponds to a requirement that all proarrows be "companionable".

Using companion structure, we can interpret a form of covariant Kan composition in our double-categorical universe models. For example, we can fill the "open box"

$$\begin{tikzcd}
A \arrow[d, "f"'] & D \arrow[d, "\mathrm{id}"] \\
B \arrow[r, "N"', "|" marking] & D
\end{tikzcd}
\quad \text{with the square} \quad
\begin{tikzcd}
A \arrow[r, "\hat{f}", "|" marking] \arrow[d, "f"'] \arrow[dr, phantom, "\cdot f \lrcorner"] & B \arrow[r, "N", "|" marking] \arrow[d, dashed, "\mathrm{id}" description] \arrow[dr, phantom, "\mathrm{id}"] & D \arrow[d, "\mathrm{id}"] \\
B \arrow[r, "U"', "|" marking] & B \arrow[r, "N"', "|" marking] & D
\end{tikzcd}$$

However, in order to fill an open box of the form

$$\begin{tikzcd}
A \arrow[d, "f"'] & C \arrow[d, "g"] \\
B \arrow[r, "N"', "|" marking] & D
\end{tikzcd}$$

we would need a proarrow corresponding to the reverse of the arrow g. Such a proarrow is known as a *conjoint*, and has properties dual to those of companions. It happens that a companion to a left adjoint, if it exists, is necessarily a conjoint. This tells us when a function gives rise to a path in the reverse direction. We may recover symmetrical paths in a universe by designating as companionable the arrows that participate in an adjoint equivalence, because the arrows of an adjoint equivalence are each left adjoint to the other.

References

[1] Robert Dawson and Robert Pare. "General Associativity and General Composition for Double Categories". In: *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 31.1 (1993), pp. 57–79.

[2] Marco Grandis and Robert Pare. "Adjoint for Double Categories". In: *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 45.3 (2004), pp. 193–240.

[3] Emily Riehl and Michael Shulman. "A type theory for synthetic ∞-categories". In: *Higher Structures* 1.1 (2017).

Compositional Game Theory in Type Theory

Fredrik Nordvall Forsberg

University of Strathclyde, Glasgow, UK
`fredrik.nordvall-forsberg@strath.ac.uk`

Introduction Game theory is the mathematical theory of rational agents trying to make optimal decisions, as in the famous example of the Prisoner's Dilemma, where two prisoners have to decide whether to give each other up to the authorities or not. Game theory is a major tool used in e.g. economics, political science, computer science, biology, and philosophy, but for this tool to be useful, software support is needed — it is not feasible to work with large models of games using pen and paper only. Unfortunately, computationally modelling and finding optimal decisions in large games is also computationally hard [Daskalakis et al., 2009].

Compositional game theory Compositional game theory, as introduced by Hedges [2016] (see also the work of Escardó and Oliva [e.g. 2010]), aims to tackle the problem of scalability by supporting a compositional modelling of games. This is achieved by equipping games with input/output "ports" for interacting with their environment, resulting in so-called *open games*. The ports are specified by two pairs of sets (X, S) and (Y, R), where we think of X as the history (or state) of the game, Y as the set of possible moves, R as the type of possible utilities (payoffs) in the game, and S as the type of utilities passed on to the environment. A game is then given by a set of valid strategies, functions explaining how strategies give rise to moves, and how strategies convert utility into coutility (i.e. utility for the environment), as well as a function specifying which strategies are optimal for each utility function $u : Y \to R$ — the equilibria of the game (so called since they often coincide with equilibrium concepts in concrete games):

Definition 1. Let X, Y, S, R : Set. An *open game* $\mathcal{G} = (\Sigma_\mathcal{G}, P_\mathcal{G}, C_\mathcal{G}, E_\mathcal{G}) : (X, S) \to (Y, R)$ consists of:

- a set $\Sigma_\mathcal{G} : \mathsf{Set}$, called the set of *strategy profiles* of $\mathcal{G}$,
- a function $P_\mathcal{G} : X \to \Sigma_\mathcal{G} \to Y$, called the *play function* of $\mathcal{G}$,
- a function $C_\mathcal{G} : X \to \Sigma_\mathcal{G} \to R \to S$, called the *coutility function* of $\mathcal{G}$, and
- a function $E_\mathcal{G} : X \to (Y \to R) \to \mathscr{P}(\Sigma_\mathcal{G})$, called the *equilibrium function* of $\mathcal{G}$.

We represent the power set $\mathscr{P}(A)$ in type theory by $\mathscr{P}(A) = A \to \mathsf{Prop}$ (a proof-relevant version $\mathscr{P}(A) = A \to \mathsf{Set}$ also seems possible). Using the "type information" present in a game $\mathcal{G} : (X, S) \to (Y, R)$, we can compose it with other games, e.g. given $\mathcal{G}' : (X', S') \to (Y', R')$ we can construct the game $\mathcal{G} \otimes \mathcal{G}' : (X \times X', S \times S') \to (Y \times Y', R \times R')$ where $\mathcal{G}$ and $\mathcal{G}'$ are played in parallel, and given $\mathcal{H} : (Y, R) \to (Z, T)$ with a *matching* interface (Y, R), we can construct the game $\mathcal{H} \circ \mathcal{G} : (X, S) \to (Z, T)$ where $\mathcal{G}$ and then $\mathcal{H}$ are played sequentially. In this way, larger games can be modularly built from smaller, well-understood games, instead of being constructed monolithically from scratch each time — for instance the Prisoner's Dilemma game arises as the parallel composition of two simple Prisoner games. The situation can be summed up as follows:

Theorem 2 (Ghani et al. [2018a]). *The collection of pairs of sets, with open games $\mathcal{G} : (X, S) \to (Y, R)$ as morphisms, forms a symmetric monoidal category* Game. *There is a faithful functor* $\mathsf{Set} \times \mathsf{Set}^{\mathsf{op}} \to \mathsf{Game}$ *embedding a pair of functions (f, g) as the play and coutility functions of a strategically trivial game.*

Notice how the opposite category $\mathsf{Set}^{\mathsf{op}}$ takes the "contravariance" of coutilities into account. This model also supports many other operations, such as infinite repetition [Ghani et al., 2018b].

Dependently typed open games However, the model described above is not flexible enough to accurately describe certain games; most notably, strategies are required to be playable in all states (by the type of the play function $P_{\mathcal{G}} : X \to \Sigma_{\mathcal{G}} \to Y$), and utility functions must be ready to assign a payoff from the same set R to all moves. As a concrete example, this precludes the construction of a well-behaved *external choice* operator $\oplus$ on games, where the state set of the game $\mathcal{G} \oplus \mathcal{G}'$ is the disjoint union of the state sets of $\mathcal{G}$ and $\mathcal{G}'$ — we would hope that this would be a coproduct of games in a natural way, but that hope is thwarted by the need to over-approximate the strategy set; in general we need a strategy from each game, but for any fixed state, a strategy from only one game would suffice. To fix this imprecision, we turn to dependent types, and replace pairs of sets in the interface of games with families of sets:

Definition 3. Let X, Y : Set, $S : X \to \mathsf{Set}$, $R : Y \to \mathsf{Set}$. A *dependently typed open game* $\mathcal{G} = (\Sigma_{\mathcal{G}}, P_{\mathcal{G}}, C_{\mathcal{G}}, E_{\mathcal{G}}) : (X, S) \to (Y, R)$ consists of:

- a family of sets $\Sigma_{\mathcal{G}} : X \to \mathsf{Set}$,
- a function $P_{\mathcal{G}} : \big(x : X\big) \to \Sigma_{\mathcal{G}}(x) \to Y$,
- a function $C_{\mathcal{G}} : \big(x : X\big) \to \big(\sigma : \Sigma_{\mathcal{G}}\big) \to R(P_{\mathcal{G}}\, x\, \sigma)) \to S(x)$, and
- a function $E_{\mathcal{G}} : \big(x : X\big) \to \big(y : Y\big) \to R(y) \to \mathscr{P}(\Sigma_{\mathcal{G}}(x))$.

We see that if S, R and $\Sigma_{\mathcal{G}}$ are all constant families — i.e. if there is no real dependency — then a dependently typed open game is exactly an ordinary open game. Again, we can construct the parallel and sequential composition of dependently typed open games, and we get:

Theorem 4. *The collection of families of sets, with dependently typed open games $\mathcal{G} : (X, S) \to (Y, R)$ as morphisms, forms a symmetric monoidal category* DGame. *There is a faithful functor* $\mathsf{Fam}(\mathsf{Set}^{\mathsf{op}}) \to \mathsf{DGame}$.

The category $\mathsf{Fam}(\mathsf{Set}^{\mathsf{op}})$ is also known as the category of containers [Abbott et al., 2005], which suggests an intriguing connection between game theory and the theory of data types.

References

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009.

Martín Hötzel Escardó and Paulo Oliva. Selection functions, bar recursion and backward induction. *Mathematical Structures in Computer Science*, 20(2):127–168, 2010.

Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. Compositional game theory. In *LICS 2018*, pages 472–481, 2018a.

Neil Ghani, Clemens Kupke, Alasdair Lambert, and Fredrik Nordvall Forsberg. A compositional treatment of iterated open games. *Theoretical Computer Science*, 741:48–57, 2018b.

Jules Hedges. *Towards compositional game theory.* PhD thesis, Queen Mary University London, 2016.

Dependable Atomicity in Type Theory

Andreas Nuyts[1] and Dominique Devriese[2]

[1] imec-DistriNet, KU Leuven, Belgium
[2] Vrije Universiteit Brussel, Belgium

Presheaf semantics [Hof97, HS97] are an excellent tool for modelling relational preservation properties of (dependent) type theory. They have been applied to parametricity (which is about preservation of relations) [AGJ14], univalent type theory (which is about preservation of equivalences) [BCH14, Hub15], directed type theory (which is about preservation of morphisms) and even combinations thereof [RS17, CH19]. Of course, after going through the endeavour of constructing a presheaf model of type theory, we want type-theoretic profit, i.e. we want internal operations that allow us to write cheap proofs of the 'free' theorems [Wad89] that follow from the preservation property concerned.

While the models for univalence, parametricity and directed type theory are all just cases of presheaf categories, approaches to internalize their results do not have an obvious common ancestor (neither historically nor mathematically). Cohen et al. [CCHM16] have used the final type extension operator Glue to prove univalence. In previous work with Vezzosi [NVD17], we used Glue and its dual, the initial type extension operator Weld, to internalize parametricity to some extent. Before, Bernardy, Coquand and Moulin [BCM15, Mou16] have internalized parametricity using completely different 'boundary filling' operators Ψ (for extending types) and Φ (for extending functions). Unfortunately, Ψ and Φ have so far only been proven sound with respect to substructural (affine-like) variables of representable types (such as the relational or homotopy interval $\mathbb{I}$). More recently, Licata et al. [LOPS18] have exploited the fact that the homotopy interval $\mathbb{I}$ is **atomic**[1] — meaning that the exponential functor $(\mathbb{I} \to \sqcup)$ has a right adjoint $\surd$ — in order to construct a universe of Kan-fibrant types from a vanilla Hofmann-Streicher universe [HS97] internally.

A failed attempt to prove parametricity of System F in ParamDTT [NVD17] using Glue and Weld, set us on a quest to figure out what is the proper way to internalize presheaf semantics. A comparison of the expressive power of Glue, Weld, Φ, Ψ and a few additional operators, revealed that Φ cannot be implemented in terms of these other operators and strongly suggested that — in this set of operators — Φ is indispensible when it comes to proving parametricity of System F [ND18]. This is an unfortunate result, as our models of parametricity with identity extension [Nuy18] are incompatible with the substructurality of interval variables required by Φ and Ψ.[2]

We propose a property that we will call *dependable atomicity* as a key notion to internalize presheaf semantics. Roughly speaking, we call a closed type $\mathbb{I}$ dependably atomic if the (potentially substructural) dependent function type former $((i : \mathbb{I}) \multimap \sqcup) : \mathrm{Ty}(\Gamma, i : \mathbb{I}) \to \mathrm{Ty}(\Gamma)$ has a right adjoint $(i \between \sqcup) : \mathrm{Ty}(\Gamma) \to \mathrm{Ty}(\Gamma, i : \mathbb{I})$ which we will call the **transpension**[3]. Dependable atomicity of $\mathbb{I}$ can be internalized using a transpension type former from which we can implement Ψ. Interestingly, this is feasible both in substructural and in cartesian settings.

All results presented below are preliminary; we are working on a proof assistant Menkar [ND19] in order to be able to trust our proofs.

Breaking down presheaf operators Let's assume we are working in MLTT with a universe of definitionally proof-irrelevant propositions φ : Prop which can be used as types and which tend to reduce to $\top$ when satisfied. Assume furthermore that we have *extension types* $A[\varphi\,?\,a]$ classifying terms of type A that become definitionally equal to a when $\varphi \equiv \top$. Under these circumstances, Moulin's Ψ-operator [Mou16] can be implemented using the transpension type and a strictness axiom as used by Orton and Pitts [OP18]. Values of the transpension type $i \between T$ can

[1]They use the word tiny, which denotes a weaker property that is equivalent in presheaf categories.
[2]Discreteness of the Π-type is essentially proven by swapping the function argument with an interval variable, but the substructural interval variables do not admit the exchange law.
[3]A sensible name would be 'dependent amazing right adjoint', as $\surd$ is sometimes called the 'amazing right adjoint', but in our opinion the name 'transpension' is more intuitive for reasons explained further.

be dependently eliminated to a restricted class of motive types, which we call **transpensive** in dimension i. Using this dependent eliminator, we can implement the operator Φ for constructing functions to transpensive types. The $\surd$ operator can be implemented as $(i : \mathbb{I}) \to i \between \sqcup$ in *cartesian* settings. Certain instances of Glue and Weld can be constructed using Φ and Ψ in a cumbersome way [ND18], but as Orton and Pitts show [OP18], Glue can already be implemented from a strictness axiom. A similar result holds for Weld, though we need an additional pushout type former for creating simple higher inductive types. Finally, a form of higher dimensional pattern matching (HDPM) which allows proving theorems such as $(\mathbb{I} \multimap A \uplus B) \to (\mathbb{I} \multimap A) \uplus (\mathbb{I} \multimap B)$ or $((i : \mathbb{I}) \multimap \mathsf{Weld}\,\{A \to (i = 0 \vee i = 1\,?\,T, f)\}) \to (\mathbb{I} \multimap A)$, becomes possible using the transpension type.

We can implement → using ↓	Ψ	Φ	$\surd$	Glue	Weld	HDPM
transpension	•	•	• (cart.)			•
dep. transp. elimination		•				
strictness axiom [OP18]	•			•	•	
pushouts along $\mathsf{snd} : \varphi \times A \to A$					•	

The transpension type If we model type theory in presheaves over a symmetric semi-cartesian base category $\mathcal{I}$ and interpret context extension with $i : \mathbb{I}$ (where $\mathbb{I} = \mathbf{y}I$ is some representable object) as a Day-convolution rather than a cartesian product (which generally requires an affine treatment of such variables), then we can soundly introduce a transpension type with the following unusual formation and introduction rules akin to rules proposed for the Φ-combinator [BV17]:

$$\frac{\Gamma, (i : \mathbb{I}) \multimap \Delta \vdash T\ \mathsf{type}}{\Gamma, i : \mathbb{I}, \Delta \vdash i \between T\ \mathsf{type}}\textsc{transp} \qquad \frac{\Gamma, (i : \mathbb{I}) \multimap \Delta \vdash t : T}{\Gamma, i : \mathbb{I}, \Delta \vdash \mathsf{merid}\ t\ i : i \between T}\textsc{merid}$$

Elimination is done using $\mathsf{unmerid} : ((i : \mathbb{I}) \multimap i \between T) \to T$, the co-unit of the adjunction $\multimap \dashv \between$. (A stronger elimination rule may be possible.) The above rules are natural in Γ and Δ, though not necessarily in the position of i in the context.

It is interesting to consider how we can construct terms of type $i \between T$. Clearly, we have $\lambda t.\lambda i.\mathsf{merid}\ t\ i : T \to (i : \mathbb{I}) \multimap i \between T$. However, assuming $\mathcal{I}$ is some cube category, how do we construct $t : 0 \between T$? The typing rule MERID doesn't cover that, but we can try to prove $(i : \mathbb{I}) \multimap (i = 0) \to i \between T$. Then the premise of MERID has an assumption $(i : \mathbb{I}) \multimap (i = 0)$ which is false. Thus (since MERID is invertible in the semantics), $0 \between T$ must be a singleton, and the same holds for $1 \between T$. We will call the respective elements $\mathsf{north} = \mathsf{merid}\ \_\ 0$ and $\mathsf{south} = \mathsf{merid}\ \_\ 1$. The transpension type is thus akin to a dependent version of the suspension type [Uni13, §6.5].

Further properties are obtained by making additional assumptions on the functor $\mathcal{I} \to \mathcal{I}/I : J \mapsto (J * I, \pi_2)$ to the slice category over the object $I \in \mathcal{I}$ that represents $\mathbb{I} = \mathbf{y}I \in \mathsf{Psh}(\mathcal{I})$. We will call I: **cancellative** if this functor is faithful, **affine** if it is full, and **connection-free** if it is essentially surjective on objects (K, φ) where $\varphi : K \to I$ is split epi.

If I is affine and cancellative, then unmerid becomes an isomorphism and the rules TRANSP and MERID become in some sense natural w.r.t. the position of the variable i in the context. If I is moreover connection-free, then all types are transpensive w.r.t. all variables $i : \mathbb{I}$ and Φ becomes sound w.r.t. such variables. If I is cancellative and $\mathcal{I}$ is cartesian, then TRANSP and MERID are typically not natural in the position of i in the context. However, since we then have the exchange rule, we may choose to always invoke these rules as though i were the first variable in the context, putting all other variables in Δ.

Transpensivity A type A is transpensive along i if it can be torn apart and reconstructed up to isomorphism using Ψ in dimension i. If and likely only if this is the case, then we can dependently eliminate $m : i \between T$ to $A\ i\ m$ by providing values of type $A\ 0\ \mathsf{north}$, $A\ 1\ \mathsf{south}$ and for every $t : T$ a path $(i : \mathbb{I}) \multimap A\ i\ (\mathsf{merid}\ t\ i)$ connecting them. The transpension type $i \between T$ itself is transpensive, and we expect that the property is respected by most type formers (though likely not by the universe). Thus, we hope that even in a cartesian setting, we can prove that all System F types are transpensive and then use Φ to prove parametricity of System F.

Acknowledgements We want to thank Jean-Philippe Bernardy, Dan Licata and Andrea Vezzosi for inspiring discussions on the subject. Andreas Nuyts holds a Ph.D. Fellowship from the Research Foundation - Flanders (FWO).

References

[AGJ14] Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *Principles of Programming Languages*, 2014. doi:10.1145/2535838.2535852.

[BCH14] Marc Bezem, Thierry Coquand, and Simon Huber. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, Dagstuhl, Germany, 2014. URL: http://drops.dagstuhl.de/opus/volltexte/2014/4628, doi:10.4230/LIPIcs.TYPES.2013.107.

[BCM15] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electron. Notes in Theor. Comput. Sci.*, 319:67 – 82, 2015. doi:http://dx.doi.org/10.1016/j.entcs.2015.12.006.

[BV17] Jean-Philippe Bernardy and Andrea Vezzosi. Parametric application. Private communication, 2017.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016. URL: http://arxiv.org/abs/1611.02108.

[CH19] Evan Cavallo and Robert Harper. Parametric cubical type theory. *CoRR*, abs/1901.00489, 2019. arXiv:1901.00489.

[Hof97] Martin Hofmann. *Syntax and semantics of dependent types*, chapter 4, pages 13–54. Springer London, London, 1997. doi:10.1007/978-1-4471-0963-1_2.

[HS97] Martin Hofmann and Thomas Streicher. Lifting grothendieck universes. Unpublished note, 1997.

[Hub15] Simon Huber. A model of type theory in cubical sets. Licentiate's thesis, University of Gothenburg, Sweden, 2015.

[LOPS18] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal universes in models of homotopy type theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 22:1–22:17, 2018. URL: https://doi.org/10.4230/LIPIcs.FSCD.2018.22, doi:10.4230/LIPIcs.FSCD.2018.22.

[Mou16] Guilhem Moulin. *Internalizing Parametricity.* PhD thesis, Chalmers University of Technology, Sweden, 2016.

[ND18] Andreas Nuyts and Dominique Devriese. Internalizing Presheaf Semantics: Charting the Design Space. In *Workshop on Homotopy Type Theory / Univalent Foundations*, 2018. URL: https://hott-uf.github.io/2018/abstracts/HoTTUF18_paper_1.pdf.

[ND19] Andreas Nuyts and Dominique Devriese. Menkar: Towards a multimode presheaf proof assistant. In *TYPES*, 2019.

[Nuy18] Andreas Nuyts. Presheaf models of relational modalities in dependent type theory. *CoRR*, abs/1805.08684, 2018. arXiv:1805.08684.

[NVD17] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *PACMPL*, 1(ICFP):32:1–32:29, 2017. URL: http://doi.acm.org/10.1145/3110276, doi:10.1145/3110276.

[OP18] Ian Orton and Andrew M. Pitts. Axioms for modelling cubical type theory in a topos. *Logical Methods in Computer Science*, 14(4), 2018. URL: https://doi.org/10.23638/LMCS-14(4:23)2018, doi:10.23638/LMCS-14(4:23)2018.

[RS17] E. Riehl and M. Shulman. A type theory for synthetic ∞-categories. *ArXiv e-prints*, May 2017. arXiv:1705.07442.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, IAS, 2013.

[Wad89] Philip Wadler. Theorems for free! In *FPCA '89*, pages 347–359, New York, NY, USA, 1989. ACM. doi:10.1145/99370.99404.

Menkar: Towards a Multimode Presheaf Proof Assistant

Andreas Nuyts[1] and Dominique Devriese[2]

[1] imec-DistriNet, KU Leuven, Belgium
[2] Vrije Universiteit Brussel, Belgium

Recently, a number of extensions to dependent type theory have been proposed or refined that are not easily added to pre-existing proof assistants as they severely impact the nature and manipulation of type-theoretic judgements and therefore require principled implementation choices from the start. For this reason, we have started the implementation in Haskell of a novel proof assistant Menkar[1] with the aim of supporting features that are relevant to studying modal and multimode type theory as well as type-systems based on the internalization of properties of a presheaf model. We take a pragmatic approach, looking for a compromise between soundness, user-friendliness, ease of implementation, and flexibility with respect to foreseen and unforeseen modifications of the type system. Unlike projects such as NuPRL and Cedille, we do not intend to demonstrate an alternative view on type theory. Instead, we keep Menkar very Agda-like and merely aim for an implementation of some recently proposed features. Below, we discuss how we (plan to) handle each of the desired features. We conclude by listing possible applications.

Modal type theory In modal type theory, all functions and all variables are annotated with a *modality* describing the behaviour of the dependency. Applications include: modal logic (eponymously) [PD01], variance of functors [Abe06, Abe08, LH11], intensionality vs. extensionality [Pfe01], irrelevance [Pfe01, Miq01, BB08, MS08, Ree03, AS12, AVW17, ND18], shape-irrelevance [AVW17, ND18], parametricity [NVD17], axiomatic cohesion [LS16] and globality [LOPS18]. Pfenning and Abel [Pfe01, Abe06, Abe08] have gradually developed a treatment in terms of an ordered monoid where left multiplication $\mu \circ \sqcup$ has a left adjoint (a Galois connection) $\mu \setminus \sqcup$ which we call *left division*. When type-checking a term $\Gamma \vdash t : T$, μ-modal subterms are type-checked in context $\mu \setminus \Gamma$, which is obtained by applying $\mu \setminus \sqcup$ to the modalities of all variables in Γ. Agda supports modalities for irrelevance and shape-irrelevance based on the ordered monoid approach and Vezzosi has made use of this to extend Agda with support for a global (a.k.a. crisp/flat) modality in `agda-flat` and with support for parametric modalities in `agda-parametric`.[2]

Multimode type theory Sometimes, the set of available modalities μ for functions $(\mu \mid x : A) \to B$ depends on the types A and B. For example, in previous work [ND18] we developed a type system which we will refer to as **RelDTT**, in which functions from $\mathbb{N}$ to Bool are either ad hoc or irrelevant, whereas functions from the universe to Bool can also be parametric and functions from $\mathbb{N}$ to the universe can also be shape-irrelevant. In System F, there is always at most one modality applicable, but it is not always the same: functions between types are always ad hoc, while functions from kinds to types are always parametric. Recently, Licata and Shulman have explained these phenomena by moving from an ordered monoid to a 2-category, whose objects are called **modes** and whose morphisms serve as modalities. If there happens to be only a single mode, then we are essentially back in the ordered monoid setting. In case there are or may be multiple modes, we speak of multimode type theory, which is thus a generalization of modal type theory. Here, one assigns a mode to every type, and the modality of a function must match the domain and codomain modes. For System F, we could have 2 modes: `data` for types classifying data and `type` for kinds classifying types. The modes of RelDTT are called $-1, 0, 1, 2, \ldots$ but could be read as `proof`, `data`, `type`, `kind`, etc. In Menkar, we aim to support arbitrary multimode type systems. Every declaration and every variable in Menkar is annotated (implicitly or explicitly) with a mode matching its type, and a modality that lifts it to the mode of the enclosing module or context. When type-checking a declaration, all other declarations that are in scope because

[1]`http://github.com/anuyts/menkar`
[2]See the `flat` and `parametric` branches at `https://github.com/agda/agda`.

they are part of enclosing modules, are simply added to the context. This ensures that they too are subject to the correct left divisions as we move into modal subterms.

Internal mode and modality polymorphism RelDTT [ND18] has infinite sets of modes and modalities. For this reason, Menkar support for internal mode and modality polymorphism is more than desirable. For most type systems, this will be stretching the semantics, but in general we intend to parametrize Π- and Σ-types as well as the universe with mode and/or modality arguments, on which they depend crisply [LOPS18]. The most striking consequence of modality polymorphism is that it becomes impossible to compute $\mu \backslash \Gamma$ by dividing every individual modality in Γ, because μ may depend on all variables in Γ. Hence, in our implementation, left division is a constructor of the context type. When type-checking a variable, it is checked that the variable's modality is less than the composite of all modalities it has been divided by.

Parametric Tarski-universes Our type systems for parametricity [NVD17, ND18] share the remarkable property that the type T of a term $\Gamma \vdash t : T$ is checked in context $\mathbf{par} \setminus \Gamma$, i.e. divided by parametricity. Fortunately, this does not require any heavyweight language support. Instead, the language provides a (typically non-fibrant) universe `UniHS` (typically modelled by the Hofmann-Streicher universe [HS97] available in all presheaf models) such that $\Gamma \vdash t : T$ requires $\Gamma \vdash T : \texttt{UniHS}$. The fibrant universe is then a different type `Uni` equipped with a parametric function $\texttt{El} : (\mathbf{par} \mid \texttt{Uni}) \to \texttt{UniHS}$.

Transpension and affineness In parallel work [ND19] we propose a novel *transpension* type as key to internalizing presheaf semantics. This type requires unusual context manipulation, including the disappearance of variables and universal quantification of other variables. We expect that these operations can be captured using a variable-indexed modality system with modalities expressing 'fresh for i', 'for all i' and 'transpend over i'. We expect these same modalities can be used to capture semantically related phenomena related to the substructural affine-like interval variables used by Bernardy, Coquand and Moulin [BCM15, Mou16].

Proving fibrancy internally As mentioned before, Menkar provides a type `UniHS` which is in general non-fibrant. Indeed, our intention is to model Menkar in the default CwF on an arbitrary presheaf category [Hof97] and not in a CwF that restricts to fibrant types. Instead, we want to prove fibrancy internally, avoiding unreadable technical reports such as [Nuy18]. This should often be possible using the transpension type, which is more general than the $\surd$-operator used by Licata et al. [LOPS18], to internalize CCHM fibrancy [CCHM16]. The fibrancy proofs might be made available in a practical way using instance arguments [DP11].

Relatedness-checking RelDTT [ND18] relies on the notion of *judgemental relatedness* [Vez17]. We have implemented the core of a relatedness-checker for Menkar, though we are not sure how to use it. In RelDTT, all types are fibrant (i.e. discrete) meaning that equality coincides with 0-relatedness. Thus, we need not distinguish between judgemental equality and judgemental 0-relatedness and can seamlessly move from conversion-checking to relatedness-checking, ultimately ignoring irrelevant subterms. However, in Menkar, types are non-fibrant unless *proven* otherwise; hence, a definitional mechanism cannot rely on fibrancy.

Interestingly, using instance arguments it may also be possible to implement a relatedness-checker *within* Menkar, which produces propositional evidence. To do so, we would provide an instance for every term constructor of the language. In order to avoid that the instance for e.g. application fires always, we should be able to restrict the instance to neutral function terms.

Applications Our own motivation to start the work on Menkar is to obtain an implementation of RelDTT and of the transpension type, as well as to research a directed version of RelDTT. However, we believe that Menkar's features are also valuable for studying cubical HoTT, as well as guarded type theory including clock-irrelevance [BGC+16] and time warps [Gua18].

Acknowledgements Thanks to Andrea Vezzosi for discussions about type-checking a type-system with extents, and relatedness-checking, and to Andreas Abel for related discussions. Andreas Nuyts holds a Ph.D. Fellowship from the Research Foundation - Flanders (FWO).

References

[Abe06] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[Abe08] Andreas Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 18(5):797–822, 2008. URL: `https://doi.org/10.1017/S0960129508006853`, `doi:10.1017/S0960129508006853`.

[AS12] Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1):1–36, 2012. TYPES'10 special issue. `doi:http://dx.doi.org/10.2168/LMCS-8(1:29)2012`.

[AVW17] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. Normalization by evaluation for sized dependent types. *Proc. ACM Program. Lang.*, 1(ICFP):33:1–33:30, August 2017. URL: `http://doi.acm.org/10.1145/3110277`, `doi:10.1145/3110277`.

[BB08] Bruno Barras and Bruno Bernardo. *The Implicit Calculus of Constructions as a Programming Language with Dependent Types*, pages 365–379. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. `doi:10.1007/978-3-540-78499-9_26`.

[BCM15] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electron. Notes in Theor. Comput. Sci.*, 319:67 – 82, 2015. `doi:http://dx.doi.org/10.1016/j.entcs.2015.12.006`.

[BGC$^+$16] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In *FOSSACS '16*, 2016. `doi:10.1007/978-3-662-49630-5\_2`.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016. URL: `http://arxiv.org/abs/1611.02108`.

[DP11] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in Agda. In *16th International Conference on Functional Programming*, pages 143–155. ACM, 2011.

[Gua18] Adrien Guatto. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 482–491, 2018. URL: `https://doi.org/10.1145/3209108.3209148`, `doi:10.1145/3209108.3209148`.

[Hof97] Martin Hofmann. *Syntax and semantics of dependent types*, chapter 4, pages 13–54. Springer London, London, 1997. `doi:10.1007/978-1-4471-0963-1_2`.

[HS97] Martin Hofmann and Thomas Streicher. Lifting grothendieck universes. Unpublished note, 1997.

[LH11] Daniel R. Licata and Robert Harper. 2-dimensional directed type theory. *Electr. Notes Theor. Comput. Sci.*, 276:263–289, 2011. URL: `https://doi.org/10.1016/j.entcs.2011.09.026`, `doi:10.1016/j.entcs.2011.09.026`.

[LOPS18] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal universes in models of homotopy type theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 22:1–22:17, 2018. URL: `https://doi.org/10.4230/LIPIcs.FSCD.2018.22`, `doi:10.4230/LIPIcs.FSCD.2018.22`.

[LS16] Daniel R. Licata and Michael Shulman. *Adjoint Logic with a 2-Category of Modes*, pages 219–235. Springer International Publishing, 2016. `doi:10.1007/978-3-319-27683-0_16`.

[Miq01] Alexandre Miquel. The implicit calculus of constructions. In *TLCA*, pages 344–359, 2001. URL: `https://doi.org/10.1007/3-540-45413-6_27`, `doi:10.1007/3-540-45413-6_27`.

[Mou16] Guilhem Moulin. *Internalizing Parametricity.* PhD thesis, Chalmers University of Technology, Sweden, 2016.

[MS08] Nathan Mishra-Linger and Tim Sheard. *Erasure and Polymorphism in Pure Type Systems*, pages 350–364. 2008. URL: https://doi.org/10.1007/978-3-540-78499-9_25, doi:10.1007/978-3-540-78499-9_25.

[ND18] Andreas Nuyts and Dominique Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 779–788, 2018. URL: https://doi.org/10.1145/3209108.3209119, doi:10.1145/3209108.3209119.

[ND19] Andreas Nuyts and Dominique Devriese. Dependable atomicity in type theory. In *TYPES*, 2019.

[Nuy18] Andreas Nuyts. Presheaf models of relational modalities in dependent type theory. *CoRR*, abs/1805.08684, 2018. arXiv:1805.08684.

[NVD17] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *PACMPL*, 1(ICFP):32:1–32:29, 2017. URL: http://doi.acm.org/10.1145/3110276, doi:10.1145/3110276.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. doi:10.1017/S0960129501003322.

[Pfe01] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS '01*, pages 221–230, 2001. URL: https://doi.org/10.1109/LICS.2001.932499, doi:10.1109/LICS.2001.932499.

[Ree03] Jason Reed. Extending higher-order unification to support proof irrelevance. In *TPHOLs 2003*, pages 238–252. 2003. URL: https://doi.org/10.1007/10930755_16, doi:10.1007/10930755_16.

[Vez17] Andrea Vezzosi. Judgemental relatedness. Private communication, 2017.

Type Inhabitation in Simply Typed Lambda Calculus Parameterized by Width

Mateus de Oliveira Oliveira[1][*]

University of Bergen, Norway
mateus.oliveira@uib.no

Abstract

In the type inhabitation problem, we are given a type τ and the goal is to determine whether there exists a closed λ-term of type τ. This problem is known to be PSPACE complete when restricted to the simply typed λ-calculus. In this work, we study the type inhabitation problem in simply typed λ-calculus from the perspective of parameterized complexity theory. More specifically, we introduce a suitable notion of *width* of a type inhabitation proof, and show that one can decide the existence of a proof of width at most w in time $2^{O(w)} \cdot |\tau|$, where $|\tau|$ is the size of the input type τ. In other words, we show that the type inhabitation problem is fixed-parameter linear with respect to the width parameter.

Keywords: Simply Typed Lambda Calculus, Minimal Logic, Fixed Parameter Tractability.

1 Introduction

In the simply typed λ-calculus [2], a type τ is said to be inhabited, if there exists at least one closed λ-term of type τ. A type is empty if it is not inhabited. For instance, the type $((\alpha \to \beta) \to \alpha) \to \alpha$ encoding Pierce's law is a well known example of type that is empty. From the point of view of programming language theory, an empty type is a type that cannot be fulfilled by any program phrase.

In the *type inhabitation problem*, we are given a type τ, and the goal is to determine whether there exists a closed λ-term M of type τ. It can be shown that type inhabitation in the simply typed λ-calculus is PSPACE complete [5]. In other words, there is an algorithm that takes a suitable n-bit encoding of a given type τ, and determines using space at most $n^{O(1)}$ whether τ is inhabited. Additionally, by PSPACE-hardness, any computational problem that can be solved using polynomial space can be efficiently reduced to the type inhabitation problem. This implies that unless $P = NP = PSPACE$, type inhabitation cannot be solved in polynomial time.

In this work, we use techniques from the field of parameterized complexity theory [3] to cope with the intractability of the type inhabitation problem in the simply typed λ-calculus. In particular we introduce a new width measure for proofs of inhabitation. Subsequently, we show that the problem of determining whether a given input type τ has a proof of width at most w can be solved in time $2^{O(w)} \cdot |\tau|$, where $|\tau|$ is the size of τ. In the jargon of parameterized complexity theory, we show that the type inhabitation problem in the simply typed λ-calculus is *fixed-parameter linear* with respect to w. This result is interesting for three main reasons. First, the single-exponential dependence on the width parameter makes our algorithm practical on current computers for moderate values of width (say $w = 20$). Second, the width parameter w imposes no restriction on the actual size of the proofs, and indeed, proofs of constant width may be already much larger than the size of the input type τ itself. Therefore, our algorithm is able to determine whether a given type has a proof of width at most w without explicitly constructing a proof DAG (or proof tree), in case such a proof exists[1]. Third, if our algorithm determines that a proof of width w does not exist, then we at least have a quantitative lower bound for the complexity of our proof.

1.1 Our Approach

By reasoning in terms of the Curry-Howard isomorphism [4], one can establish a close correspondence between the simply typed λ-calculus and the implicational fragment of propositional intuitionistic logic, also known as minimal

[*]We acknowledge financial support from the Bergen Research Foundation in the context of the project *Groovy FPT*, and from the Research Council of Norway in the context of the project *Automated Theorem Proving from the Mindset of Parameterized Complexity Theory*.

[1]But such an explicit derivation can be constructed, if needed, in time linear in the size of a minimum-size proof of width w.

logic. Using this correspondence, one can associate with each type τ a minimal logic formula $P(\tau)$ in such a way that τ is inhabited if and only if the formula $P(\tau)$ has a proof Π in natural deduction.

Now, let τ be an inhabited type, and Π be a proof of $P(\tau)$. Then our next step is to define a suitable directed acyclic graph $G(\tau,\Pi)$, which encodes the input type τ, the structure of the proof Π (i.e. the proof DAG), and the interdependence between sub-terms occurring in the proof. The graph $G(\tau,\Pi)$ has one vertex v_t for each sub-term occurring in Π. This vertex v_t is labeled with the root symbol of t. Now the graph $G(\tau,\Pi)$ has a directed edge $(v_t, v_{t'})$ labeled with a number i, if and only if t is the i-th child of t'. Note that a term t may occur several times as a sub-term of the proof Π. Nevertheless, all such sub-terms will correspond to the same vertex v_t in the graph $G(\tau,\Pi)$. In addition to this graph, we let $G(\tau)$ be the sub-graph of $G(\tau,\Pi)$ induced by the vertices v_t where t ranges over the sub-terms of τ only.

The reason we choose to encode the whole proof Π as a graph is because in this way we can use the machinery of structural graph theory to define suitable width parameters which are meant to capture the ”complexity” of a proof. In particular, we choose as our parameter a width measure that we call *embedding preserving* tree-width, which has an associated notion of *embedding preserving* tree-decomposition. Such decompositions are essentially tree-decompositions of the proof graph $G(\tau,\Pi)$ which embeds a special isomorphism-invariant tree-decomposition of the graph $G(\tau)$. Our main result is formally stated in Theorem 1.

Theorem 1. *Given a type τ and a positive integer w, one can determine in time $2^{O(w)} \cdot |\tau|$ whether $P(\tau)$ has a minimum logic proof of width w. As a consequence, one can determine in time $2^{O(w)} \cdot |\tau|$ whether τ is inhabited.*

A proof sketch for Theorem 1 is as follows. We first construct an automaton $A(w)$ that accepts all tree decompositions of width at most w of graphs corresponding to legal proofs of tautologies in minimal logic. Even though the construction of the automaton $A(w)$ is quite intricate, this construction can be performed in time $2^{O(w)}$ because the property of *being the proof graph of some minimal logic tautology* can be checked locally by testing certain consistency conditions which intuitively encode the application of inference rules. Subsequently, we construct an automaton $B(\tau,w)$ that accepts all tree-decompositions of width at most w that embed the special isomorphism-invariant tree decomposition of the graph $G(\tau)$. This automaton can be constructed in time $2^{O(w)} \cdot |\tau|$. Subsequently we construct the intersection automaton $C(\tau,w) = A(w) \cap B(\tau,w)$ that accepts the language $\mathscr{L}(A(w)) \cap \mathscr{L}(B(\tau,w))$. Then, we have that the type τ is inhabited if and only if $\mathscr{L}(C(\tau,w)) \neq \emptyset$. This can be tested in time $2^{O(w)} \cdot |\tau|$.

We should note that any proposition that is provable in minimal logic has a proof whose structure is a tree (i.e. a graph of treewidth 1). Therefore, assuming that PSPACE is not equal to $DTIME[n]$ (i.e. the class of problems solvable in linear time), Theorem 1 cannot hold if one defines the width of a proof simply as the width of its structural proof DAG. In view of this, it is important to emphasize that the graph $G(\tau,\Pi)$ encodes substantially more information about the proof than the underlying proof DAG. In particular what makes our width measure interesting from an algorithmic point of view is the fact that it takes into consideration the way in which the formulas occurring during the proof share sub-formulas.

It is also important to note that while the treewidth parameter has been extensively studied in graph theory and used to provide fixed-parameter tractable algorithms for a large number of NP-hard computational problems [1], these techniques do not directly apply to our problem, which is a PSPACE hard problem. In particular, the proof of Theorem 1 uses new tools for the representation and manipulation of an exponential number of possible decompositions using concise data-structures based on automata theory. We believe that these techniques may be of independent interest.

References

[1] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *International Colloquium on Automata, Languages, and Programming*, pages 105–118. Springer, 1988.

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

[3] R. G. Downey and M. R. Fellows. *Fundamentals of parameterized complexity*, volume 4. Springer, 2013.

[4] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

[5] P. Urzyczyn. Inhabitation in typed lambda-calculi (a syntactic approach). In *International Conference on Typed Lambda Calculi and Applications*, pages 373–389. Springer, 1997.

Effects, Substitution and Induction
An Explosive *Ménage à Trois*

Pierre-Marie Pédrot[1] and Nicolas Tabareau[1]

Inria Rennes – Bretagne Atlantique, LS2N

Since the inception of dependent type theory, several people tried to apply the techniques coming from simply-typed settings to enrich it with new reasoning principles using effects, typically classical logic. The early attempts were mixed, if not outright failures. Most notably, Barthe and Uustalu showed that writing a typed CPS translation preserving dependent elimination was out of reach [1], and similarly Herbelin proved that CIC is inconsistent with computational classical logic [3].

Retrospectively, this should not have been that surprising. This incompatibility is the reflect of a very ancient issue: mixing classical logic with the axiom of choice, whose intuitionistic version is a consequence of dependent elimination, is a well-known source of foundational problems [5]. While in the literature much emphasis has been put on the particular case of classical logic, we argue here that this is an instance of a broader phenomenon, namely that side-effects are at odds with dependent type theory, in a pick two out of three conundrum. This mismatch is evocatively dubbed the *ménage à trois* and is embodied by the following theorem, which is a generalization of Herbelin's paradox.

Theorem 1 (Explosive *ménage à trois*)**.** *A type theory that features observable effects and enjoys both arbitrary substitution and dependent elimination is logically inconsistent.*

We describe more in detail the premises of this theorem hereafter, where $\star$ stands for any proof term, not necessarily unique.

Definition 1. *Substitution* is the admissibility of the following rule.

$$\frac{\Gamma, x : A \vdash \star : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \star : B\{x := u\}}$$

Definition 2. *Dependent elimination* on booleans is the admissibility of the following rule.

$$\frac{\Gamma, x : \mathbb{B} \vdash A : \square \qquad \Gamma \vdash \star : A\{x := \mathtt{true}\} \qquad \Gamma \vdash \star : A\{x := \mathtt{false}\}}{\Gamma, x : \mathbb{B} \vdash \star : A}$$

Finally, we need to express what it means for a type theory to be observably effectful. Intuitively, a type theory is pure when every term observationally behaves as a value. So a simple way to formalize what it means to be effectful is to say that there exists a boolean term which is not observationally equivalent to `true` nor `false`.

Definition 3. A type theory is *observably effectful* if there exists a closed term $\vdash t : \mathbb{B}$ that is not observationally equivalent to a value, that is, there exists a context C such that $C[\mathtt{true}] \equiv \mathtt{true}$ and $C[\mathtt{false}] \equiv \mathtt{true}$, but $C[t] \equiv \mathtt{false}$, where $\equiv$ denotes definitional equality.

Proof. We define equality and empty type using the standard impredicative encoding, and we take t and C as provided by Definition 2. By dependent elimination, it holds that $x : \mathbb{B} \vdash C[x] = \mathtt{true}$. By substitution, $\vdash C[t] = \mathtt{true}$. By conversion and because $C[t] \equiv \mathtt{false}$, this implies $\vdash \mathtt{false} = \mathtt{true}$. But, by dependent elimination, we also have $\vdash \mathtt{false} = \mathtt{true} \to \bot$. $\square$

Example 1. It is possible to use `callcc` [2] to write a term `decide` : $\square \to \mathbb{B}$ that decides inhabitance of a type. Obviously, `decide` A cannot evaluate to a value in general. Such terms are called *backtracking* or *non-standard*, and are the root of Herbelin's paradox.

Facing this impossibility theorem, we briefly list possible ways out and their trade-offs.

No Effects This is the good old CIC, featuring both substitution and dependent elimination.

Call-by-value Every function can expect its argument to be a value, which explains why dependent elimination is always valid: `true` and `false` are the only non-variable boolean values. Constrastingly, substitution is now by definition restricted to values. Generalizing it to arbitrary terms is not correct if there are effectful terms, as evidenced by the requirement of a *value restriction* in most systems. Albeit not strictly speaking dependent type theory, this is the path followed by PML [4].

Call-by-name In this setting, substitution always holds by construction. However, as already noticed in [6], dependent elimination is now lost in general. If there are effectful terms, knowing the behaviour of a predicate on boolean *values* is not enough to know the behaviour of the predicate in general, as there is a desynchronization between effects performed in the term and effects performed in the type during pattern-matching. This is what BTT [6] is all about.

Boom It is possible to satisfy the premises of the theorem, at the expense of consistency. The exceptional type theory [7] is such an instance. While seemingly concerning at first, one can argue that this is a paradigm shift from a dependent *type theory* to a dependently-typed *programming language*, where consistency is not relevant.

We will give more in-depth insights about these paradigms, and advocate for an encompassing theory called ∂CBPV [8]. This is a generalization of call-by-push-value to dependent types allowing for a uniform setting in which describe these effectful theories.

References

[1] G. Barthe and T. Uustalu. Cps translating inductive and coinductive types. In *Proceedings of Partial Evaluation and Semantics-based Program Manipulation*, pages 131–142. ACM, 2002.

[2] T. Griffin. A formulae-as-types notion of control. In *Seventeenth Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 47–58, 1990.

[3] H. Herbelin. On the degeneracy of sigma-types in presence of computational classical logic. In P. Urzyczyn, editor, *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2005.

[4] R. Lepigre. A classical realizability model for a semantical value restriction. In *25th European Symposium on Programming, ESOP 2016*, pages 476–502, 2016.

[5] P. Martin-Löf. 100 Years of Zermelo's Axiom of Choice: What was the problem with it? *Comput. J.*, 49(3):345–350, 2006.

[6] P.-M. Pédrot and N. Tabareau. An effectful way to eliminate addiction to dependence. In *32nd Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[7] P.-M. Pédrot and N. Tabareau. Failure is Not an Option – An Exceptional Type Theory. In *27th European Symposium on Programming, ESOP 2018*, pages 245–271, 2018.

[8] P.-M. Pédrot and N. Tabareau. The Fire Triangle. Draft at `https://www.pédrot.fr/articles/dcbpv.pdf`, 2019.

Coherence for symmetric monoidal groupoids in HoTT/UF

Stefano Piceghello

University of Bergen, Norway
stefano.piceghello@uib.no

We aim to produce a formalized proof of coherence for symmetric monoidal groupoids in HoTT, using equivalent constructions of a free symmetric monoidal groupoid that highlight different properties: commutativity of coherence diagrams, normal forms of symmetric monoidal expressions, and invariance under symmetries.

Our analysis starts from monoidal structures without symmetries. The proof of a statement for coherence for monoidal categories – namely, "in a free monoidal category, every diagram commutes" – in intensional MLTT was formalized in ALF in [1]; in there, a category consists of a set of objects and a family of Hom-setoids. A similar result can be obtained in HoTT, where we choose instead to employ the built-in higher groupoid structure of types to represent categories with all arrows invertible (i.e., groupoids), using the correspondence between objects and terms, invertible arrows and paths, and commutative diagrams and 2-paths. This set-up is sufficient to express the same statement for coherence, since (a.) all arrows in a free monoidal category are products of instances of the natural isomorphisms defining the monoidal structure, and hence they are invertible; and (b.) coherence is achieved by means of strong monoidal functors. A monoidal groupoid is then a 1-type endowed with a monoidal structure, and coherence can be formulated as follows: a free monoidal 1-type is monoidally equivalent to a monoidal 0-type.

In order to explicitly describe the free monoidal 1-type on a type X of generators, we use higher inductive types (HITs) as in [2]. We define the recursive HIT $F(X)$ with 0-constructors for the objects of the groupoid (a unit, the inclusion of the generators, and a monoidal product), 1-constructors for associativity of the monoidal product and the unit laws, 2-constructors for the coherence diagrams, and a 1-truncation. A normalization of the monoidal expressions in $F(X)$ is then used to show that this type is monoidally equivalent to the type $\mathrm{list}(X)$ of lists over X, with list concatenation as the monoidal product. This is then easily proven to be a 0-type whenever X is.

$$\begin{aligned}
F(X) ::= {} & e : F(X) \quad \mid g : X \to F(X) \quad \mid \otimes : F(X) \to F(X) \to F(X) \\
& \mid \alpha : (a,\, b,\, c : F(X)) \to (a \otimes b) \otimes c = a \otimes (b \otimes c) \quad \mid \lambda : (b : F(X)) \to e \otimes b = b \\
& \mid \rho : (a : F(X)) \to a \otimes e = a \quad \mid \tau : (a,\, b : F(X)) \to (\rho_a \otimes 1_b) \cdot \alpha_{a,e,b} = (1_a \otimes \lambda_b) \\
& \mid \pi : (a,\, b,\, c,\, d : F(X)) \to (\alpha_{a,b,c} \otimes 1_d) \cdot \alpha_{a,b\otimes c,d} \cdot (1_a \otimes \alpha_{b,c,d}) = \alpha_{a\otimes b,c,d} \cdot \alpha_{a,b,c\otimes d} \\
& \mid \mathrm{trunc} : \mathrm{IsTrunc}\ 1\ F(X)
\end{aligned}$$

Though conceptually similar to [1], our HoTT-based implementation presents important features of their own. First of all, the elimination principle of $F(X)$ guarantees that this type really represents a free monoidal groupoid, in the precise sense that the construction is left-adjoint to the forgetful functor to types (this one realized by the first projection out of a Σ-type). Secondly, in [1] the normalizing functor from monoidal expressions to lists factors through endomorphisms of $\mathrm{list}(X)$, where associativity and the unit laws hold definitionally. While appropriate for the task, this method does not generalize to other coherence theorems (e.g. when symmetry is part of the structure), so we choose to adopt a more straightforward approach, by mapping the monoidal product directly to list concatenation. Finally, as the coherence morphisms and diagrams rest on identity types, the resulting proof of a monoidal equivalence $F(X) \simeq \mathrm{list}(X)$ is much shorter than the one in [1] (performed by induction on the arrows of the category): all the cases with an analogue in the groupoid structure of identity types – inverses, composition, product of arrows – are redundant in our proof. The trade-off for this approach is its intrinsic specificity to structures with invertible morphisms only.

We then investigate symmetric monoidality. A free symmetric monoidal 1-type $FS(X)$ defined similarly to $F(X)$ can be proven to be equivalent, via a strong symmetric monoidal functor, to another HIT $\mathrm{slist}(X)$ defined with the 0-constructors of lists, 1-constructors for transpositions of two adjacent elements in a list, 2-constructors for the relations between the transpositions, matching those in the presentation of the symmetric groups Σ_n,

$$\Sigma_n := \frac{(a_1, \dots, a_{n-1})}{\begin{array}{c} a_i^2 = 1, \qquad (a_i a_{i+1})^3 = 1, \\ a_i a_j = a_j a_i \ \text{ for } |i-j| \geq 2 \end{array}} \tag{1}$$

and a 1-truncation. In contrast to $\mathrm{list}(X)$, the type $\mathrm{slist}(X)$ (and hence $FS(X)$) is, in general, not a 0-type, as indeed not every diagram in a free symmetric monoidal groupoid commutes.

While $\mathrm{slist}(X)$ is essentially meant to represent a "free permutative 1-type", i.e. a free symmetric monoidal 1-type in which associativity and the unitors are strict, our framework does not actually allow to express strictness of a monoidal structure, so coherence cannot be concluded by the established equivalence alone. A proof of coherence would instead entail showing that the connected components of $\mathrm{slist}(X)$ corresponding to lists with no repetitions are contractible. We believe that this can be attained in the following way: first of all, by exhibiting a symmetric monoidal equivalence

$$\mathrm{slist}(X) \simeq \sum_{n:\mathrm{nat}} \sum_{A:B\Sigma_n} (A \to X), \tag{2}$$

where $B\Sigma_n$ is the classifying space of the symmetric group Σ_n, and then by observing that

$$\sum_{n:\mathrm{nat}} \sum_{A:B\Sigma_n} (A \hookrightarrow X),$$

i.e. the subtype selecting symmetric monoidal expressions with no repetitions, is a 0-type. The equivalence in (2) should follow from the equivalence between the description of $B\Sigma_n$ as

$$B\Sigma :\equiv (n : \mathrm{nat}) \to \sum_{Y:\mathcal{U}} \|Y \simeq \mathrm{Fin}(n)\|_{-1}$$

and the family, indexed by nat, of deloopings of Σ_n. This can be defined as a family of 1-truncated HITs, with a basepoint in the first type in the family, an inclusion of each type into the next one, and a new loop at each type, satisfying the relations defining Σ_n as in (1).

The last equivalence and the one in (2) are, at the present time, yet to be formalized. All other claims have been formalized in Coq using the HoTT library[1].

References

[1] Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In *Types for Proofs and Programs*, pages 47–61. Springer Berlin Heidelberg, 1996.

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

[1]https://github.com/HoTT/HoTT

Twisted Cubes via Graph Morphisms

Gun Pinyo and Nicolai Kraus

University of Nottingham, Eötvös Loránd University

Short description. *Twisted cubes* are a variation of a cube category that has previously been used to model homotopy type theory [1]. Here, we describe how twisted cubes are constructed and what their properties are. The talk is based on our preprint [5].

General Motivation. Intensional Martin-Löf type theory admits models where types are interpreted as groupoids (categories where every morphism is invertible) and even higher groupoids. The latter view is particularly important to explain *homotopy* type theory. One way of approaching higher groupoids is via presheaves on cube categories with certain filling conditions, and this is how Bezem, Coquand, and Huber [1] have built an important model of homotopy type theory.

$$\begin{tikzcd}
y \arrow[r, dashed] & w \\
x \arrow[u, "p"] \arrow[r, "q"] & z \arrow[u, "r"']
\end{tikzcd}$$

The basic idea is as follows: a type is modeled as a cubical set (think of a collection of cubes of various dimensions). Points are the elements of the type, lines are the equalities, squares are the equalities between equalities, and so on. The filling condition says that, whenever we have a partial cube satisfying some properties, it can be completed (this corresponds to the Kan filling condition of horns for simplicial sets). For example, a "partial square" could be given by x, y, z, w and p, q, r as shown on the left, where we think of x, y, z, w as elements and p, q, r as equalities. The filling then tells us that there is another line, namely the dashed one, as shown in the picture; we think of it as the composition of the other three (the inner part of the square would be the evidence that it is indeed the composition).

The direction of the arrows in the picture above is determined by the concrete cube category that is used. We see that the three arrows in the above diagram cannot really be composed directly, since p goes into the wrong direction. Since equality in (homotopy) type theory is invertible, this is not an issue.

However, the idea to create a type theory where equality is not necessarily invertible is subject to current research as well, and goes under the name *directed type theory* [4, 7, 3]. In this case, the above observation is problematic. If z and w are simply x, and q and r are identities (i.e. reflexivity on x), then the composition gives us an inverse of p which should not necessarily exist. To remedy this, in the case of a square, we swap the direction of the left vertical arrow p and call it a *twisted square*. This can be generalised and gives rise to *twisted n-cubes* by recursion: To construct a twisted $(n+1)$-cube, we multiply a twisted n-cube with a directed interval (in the same way as a standard $(n+1)$-cube can be constructed from a standard n-cube), and invert everything at the first endpoint of the interval. This ensures that every face of a twisted $(n+1)$-cube is a twisted n-cube.

Technical Details. To make the above idea precise, we represent a twisted n-cube as a directed graph without parallel edges, T_n. This graph is defined recursively using $\mathsf{tw\text{-}iter}$, a function that takes any graph and "thickens and inverts at the source":

$$\begin{aligned}
(\mathsf{tw\text{-}iter}\ (V,E))_{\mathsf{nodes}} &:\equiv \{0,1\} \times V & T_0 &:\equiv (\{\epsilon\},\ \{(\epsilon,\epsilon)\}) \\
(\mathsf{tw\text{-}iter}\ (V,E))_{\mathsf{edges}} &:\equiv \{\ (\ (0,\ t),\ (0,s)\ )\ \mid\ (s,t) : E\} & T_{n+1} &:\equiv \mathsf{tw\text{-}iter}\ (T_n) \\
&\cup \{\ (\ (1,\ s),\ (1,t)\ )\ \mid\ (s,t) : E\} \\
&\cup \{\ (\ (0,\ v),\ (1,v)\ )\ \mid\ v : V\}.
\end{aligned} \tag{1}$$

One interesting feature of T_n is a connection to the (2^n-1)-simplex, witnessed by a unique Hamiltonian path through T_n. T_3 is shown on the right, with the Hamiltonian path drawn using thicker arrows. This also implies that the transitive closure of T_n is a total linear order, something that happens for simplices but not for cubes. The sketch of proof is that the third clause in the definition of edges of tw-iter links the biggest node in the first copy (which was the smallest node before it got inverted) to the smallest node in the second copy.

$$\begin{tikzcd}
{} & 011 \arrow[rr] \arrow[dl] \arrow[dd] & & 111 \\
010 \arrow[rr] \arrow[dd] & & 110 \arrow[ur] & \\
& 001 \arrow[rr] & & 101 \arrow[uu] \arrow[dl] \\
000 \arrow[ur] \arrow[rr] & & 100 \arrow[uu] &
\end{tikzcd}$$

The category of twisted cubes that we consider, denoted by $\bowtie_{\mathsf{grp}}$, has natural numbers as objects, and morphisms from m to n are graph homomorphisms from T_m to T_n.

We further add the condition to $\bowtie_{\mathsf{grp}}$ that dimensions are preserved, i.e. we only consider graph homomorphisms $f : T_m \to T_n$ such that, if e_1, e_2 are edges of T_m that go into the same direction, $f(e_1)$ and $f(e_2)$ also go into the same direction. This defines a subcategory that we denote by $\bowtie_{\mathsf{dim}}$. This subcategory has the same objects as, but fewer morphisms than $\bowtie_{\mathsf{grp}}$; and this restriction essentially excludes connections. To show why the definition of $\bowtie_{\mathsf{dim}}$ makes sense, we define a category for the standard cubes counterpart called $\square_{\mathsf{dim}}$ and prove in our preprint [5] that $\square_{\mathsf{dim}}$ is isomorphic to the opposite of the category of cubes used by Bezem, Coquand, and Huber [1]. $\square_{\mathsf{dim}}$ is essentially the same as $\bowtie_{\mathsf{dim}}$ but skips the "twisting" step. In other words, $\bowtie_{\mathsf{dim}}$ is the "twisted analogue" of the BCH cube category.

Other interesting features of $\bowtie_{\mathsf{dim}}$ include the observation that there is exactly one surjective morphism in $\bowtie_{\mathsf{dim}}(m,n)$ for all $m \geqslant n$ (and clearly none if $m < n$). As a consequence, the degeneracies are unique, i.e. one can only degenerate an n-cube to an $(n+1)$-cube in exactly one way, something that happens for globes but not for simplices or cubes.

Status of this work. We have defined the category of twisted cubes and proved several of its properties, details can be found in our arXiv preprint [5]. Our goal is to use it to model a version of "higher directed type theory", but we have not yet done this. Another goal is to analyse whether the setting allows for a development of higher categories in homotopy type theory: The construction given in (1) can easily be adapted to define the category of twisted semi-cubes simply by starting with $T_0 :\equiv (\{\epsilon\}, \emptyset)$. We can then consider Reedy-fibrant diagrams on this category into the universe of types, i.e. *twisted semi-cubical types*. The unique Hamiltonian path suggests that we can equip these types with an analogue of Rezk's *Segal condition* [6] similar to how it has been done in homotopy type theory (see e.g. [2]).

References

[1] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, 2014.

[2] Paolo Capriotti and Nicolai Kraus. Univalent higher categories via complete semi-segal types. *Proceedings of the ACM on Programming Languages*, 2(POPL'18):44:1–44:29, dec 2017. Full version available at https://arxiv.org/abs/1707.03693.

[3] Paige Randall North. Towards a directed homotopy type theory. *arXiv:1807.10566*, 2018.

[4] Andreas Nuyts. Towards a directed homotopy type theory based on 4 kinds of variance. Master's thesis, KU Leuven, 2015.

[5] Gun Pinyo and Nicolai Kraus. From cubes to twisted cubes via graph morphisms in type theory. *Preprint, arXiv: 1902. 10820*, 2019.

[6] Charles Rezk. A model for the homotopy theory of homotopy theory. *Trans. Amer. Math. Soc.*, 353(3):973–1007 (electronic), 2001.

[7] E. Riehl and M. Shulman. A type theory for synthetic ∞-categories. *arXiv:1705.07442*, 2017.

Towards Curry-Howard for Shared Mutable State

Pedro Rocha and Luís Caires

NOVA LINCS and Departamento de Informática,
FCT, Universidade Nova de Lisboa

Session-based concurrent computation has a firm logical foundation. In [CP10] the authors establish a tight Curry-Howard correspondence between session types and linear logic, in which types are interpreted as propositions, processes as proofs and communication as cut elimination. Within this approach, session-typed processes enjoy subject reduction, never get stuck, are confluent and always terminate. Apart from establishing the canonicity of session-based concurrency, this correspondence also provides a solid ground to explore the fruitful interplay between concurrent computation and linearity. In fact, several notions were further investigated such as logical relations, dependent types and polymorphism.

Although session types embody a notion of state in the sense of protocol evolutions, the nature of the abovementioned interpretations tells us that computation in these models is pretty much functional [TY18], while most mainstream interactive concurrent software systems not only communicate but also share and manipulate stateful resources. While linearity has been extensively explored to discipline the use of shared state, for example, in [TP10], [KTDG12, BP17], the formulation of mutable shared state within a Curry-Howard interpretation is challenging, and stands as an open problem.

In this talk we describe an approach towards a pure Curry-Howard interpretation of session-based concurrency with mutable shared state, that preserves equational reasoning over proofs by exploring internal representations of non-determinism, along the lines of [CP17]. We provide an interpretation of a conservative extension of linear logic using a fine-grained variant of the pi-calculus extended with (first class) reference cells. Each reference cell has a unique identifier and stores a server session. Cell interactions are typed by two specific dual logical exponential modalities, which are related to DiLL modalities [Ehr18]. Besides memory read and write operations, all cell usages eventually have to be released, as enforced by the cell session protocol.

Reference cells are subject to a linear usage unless they are explicitly shared by two or more processes, using co-contraction. Nondeterminism arises naturally by abstraction of the actual order of cell operations in a sharing process, where the interleaving law for process algebras (roughly $a \| b = a; b + b; a$) appears as a natural commuting conversion principle in our logic. Computations then evolve to a nondeterministic superposition of processes, which correspond to the different possible states the store can reach. Since we handle nondeterminism

$$\frac{}{x.\mathbf{cell}\langle y\rangle \vdash x : -A; \Gamma, y : \overline{A}} \qquad \frac{P \vdash \Delta; \Gamma}{x.\mathbf{free}; P \vdash \Delta, x : +A; \Gamma}$$

$$\frac{P \vdash \Delta, x : +A; \Gamma, y : A}{x.\mathbf{read}(y); P \vdash \Delta, x : +A; \Gamma} \qquad \frac{P_1 \vdash \Delta_1, y : !\overline{A}; \Gamma \qquad P_2 \vdash \Delta_2, x : +A; \Gamma}{x.\mathbf{write}(y.P_1); P_2 \vdash \Delta_1, \Delta_2, x : +A; \Gamma}$$

$$\frac{P_1 \vdash \Delta_1, x : +A; \Gamma \qquad P_2 \vdash \Delta_2, y : +A; \Gamma}{\mathbf{share}^z_{x,y}(P_1,\ P_2) \vdash \Delta_1, \Delta_2, z : +A; \Gamma} \qquad \frac{P_1 \vdash \Delta; \Gamma \qquad P_2 \vdash \Delta; \Gamma}{P_1 + P_2 \vdash \Delta; \Gamma}$$

Table 1: Typing rules for cell manipulation, sharing and nondeterminism.

explicitly [CP17, Ehr18], proof reduction is naturally confluent. Furthermore, typed processes in our calculus satisfy both progress and type preservation. We will also discuss future work directions.

References

[BP17] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):37, 2017.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory*, pages 222–236. Springer, 2010.

[CP17] Luís Caires and Jorge A Pérez. Linearity, control effects, and behavioral types. In *European Symposium on Programming*, pages 229–259. Springer, 2017.

[Ehr18] Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28(7):995–1060, 2018.

[KTDG12] Neelakantan R Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. *ACM SIGPLAN Notices*, 47(9):41–54, 2012.

[TP10] Jesse A Tov and Riccardo Pucella. Stateful contracts for affine types. In *European Symposium on Programming*, pages 550–569. Springer, 2010.

[TY18] Bernardo Toninho and Nobuko Yoshida. On polymorphic sessions and functions. In *European Symposium on Programming*, pages 827–855. Springer, Cham, 2018.

KNOWLEDGE REPRESENTATION WITH HOTT

ANDREI RODIN (RUSSIAN ACADEMY OF SCIENCES)

While the concept of Formal Ontology already secured its central role in Knowledge Representation (KR), *Formal Epistemology* presently remains a purely philosophical subject with no direct application in KR [5]. As a result, standard formal tools used in KR such as Description Logics (DL) and Web Ontology Language (OWL) lack any *epistemic* semantics, which is an obvious miss since these tools are supposed to help us to represent *knowledge.* Formal semantics qualifies as epistemic when it supports a formal distinction between true propositions, on the one hand, and propositions such that their truthness is *known* by an epistemic agent, on the other hand. Accordingly, an epistemic semantic of logical inferences requires that a given inference not only preserves the truthness of premises but also that the truth-preservation property of a given inference is made evident to an agent (which may put strong restrictions on admissible inferential structures). Since OWL and DL use the standard truth-condition logical semantics rather than a version of proof-theoretic semantics [4] OWL and DL do not formally distinguish between those truth-preserving inferences, which are epistemically admissible and those, which are not.

Even if the concept of knowledge remains a subject of wide philosophical controversy, the very idea that knowledge of a proposition should not be identified with this proposition itself is hardly controversial. According to an influential view, an agent *knows* that P just in case (i) P is true and (ii) she *justifiedly beliefs* that P. (This view constitutes the so-called JTB theory of knowledge). While the issue of human belief belongs to human psychology and is arguably beyond the scope of theoretical KR, the epistemically-laden concept of *justification* allows for a formal treatment [1] and fully belongs to its scope. The fact that standard KR architectures do not support justificatory procedures, on the practical side, means that a regular user of KR system typically is not in a position to judge whether the "knowledge" she obtains from this system is reliable or not unless she uses some external means and tools for checking it. The present proposal aims at integrating relevant justificatory procedures into the KR architecture itself.

A justificatory procedure related to certain propositional knowledge has its formal dual in the form of verification of the corresponding procedural knowledge aka knowledge-how, that is, knowledge how to perform a given procedure. In this case the epistemic goal is not to justify a proposition but to assure that an accomplished construction has some required

Date: April 28, 2019.

Key words and phrases. Knowledge Representation and Homotopy Type theory and Justification and Verification.

properties (think of technological processes which certain desired outcomes). Since this difference in epistemic goals does not affect the basic semantics, our proposed approach applies to both these sorts of tasks.

We propose to use HoTT and its proof-theoretic semantics as a formal semantic framework for KR, which satisfies the above desiderata. The following features of HoTT motivate this choice.

(1) HoTT admits the constructive epistemically-laden proof-theoretic semantics intended by Martin-Löf's Type for MLTT [2].
(2) The new interpretation of equality in HoTT gives rise to the notion of cumulative h-hierarchy of types which, in particular, supports the distinction between propositional and higher-level types. This is the crucial feature of HoTT, which allows for representing objects (of various levels) and propositions "about" these objects within the same framework. Each such object serves as a witness/truth-maker for proposition obtained via the propositional truncation of type where the given object belongs.
(3) HoTT involves a system of formal rules, which are interpreted as logical rules at the propositional h-level and as rules for object-construction at all higher levels. This feature of HoTT, which is not available in the "flat" extensional MLTT, allows for representing various extra-logical procedures (such as material technological procedures) keeping track of the corresponding logical procedures at the propositional level of representation.

A simple example of using HoTT for representing the empirical knowledge of identity of Morning Star and Evening Star is given in my [3]. Here I interpret the observable trajectory of Venus as a path in the sense of HoTT. Other perspective applications of this approach may be less direct but the same geometrical intuition associated with HoTT can be useful in such cases too.

References

[1] S. Artemov and M. Fitting. *Justification Logic.* Cambridge University Press, 2019.
[2] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory.* Bibliopolis, 1984.
[3] A. Rodin. Venus Homotopically. *IfCoLog Journal of Logics and their Applications*, 4(4):1427–1446, 2017. Preprint: http://philsci-archive.pitt.edu/12116/.
[4] Peter Schroeder-Heister. Proof-theoretic semantics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, spring 2018 edition, 2018.
[5] Jonathan Weisberg. Formal epistemology. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, winter 2017 edition, 2017.

CONGRUENCE IN UNIVALENT TYPE THEORY

LUIS SCOCCOLA

Introduction. The main purpose of congruence closure procedures is to automate the application of basic properties of equality, such as transitivity and congruence (i.e., $x = y$ implies $f(x) = f(y)$). The only current implementation of a full congruence closure procedure for intensional type theory, Selsam & de Moura (IJCAR 2016), adds an axiom to the type theory that is inconsistent with univalence. This axiom is used when proving the congruence lemmas. We describe an approach for automatically synthesizing congruence lemmas that is compatible with univalence.

Congruence using heterogeneous equality. The main difficulty with congruence in dependent type theory is that the "obvious" congruence lemmas don't type check. For example, given $f : (a : A) \to B(a)$, the expression $\mathsf{congr}_f : (a, a' : A) \to (a = a') \to f(a) = f(a')$ doesn't type check, since $f(a) : B(a)$ whereas $f(a') : B(a')$. One way to fix this, is to use heterogeneous equalities, a weakening of McBride's "John Major equality".

Definition 1. ***Heterogeneous equality*** *is defined as an inductive family*

$$\mathsf{heq} : (A, A' : \mathcal{U}) \to A \to A' \to \mathcal{U}$$

with one constructor $\mathsf{refl}^{\mathsf{heq}} : (A : \mathcal{U}) \to (a : A) \to \mathsf{heq}(A, A, a, a)$.

One can use this to write congruence lemmas that type check, $\mathsf{congr}_f : (a, a' : A) \to \mathsf{heq}(A, A, a, a') \to \mathsf{heq}(B(a), B(a'), f(a), f(a'))$, but these cannot be proven without modifying the type theory. The solution of Selsam & de Moura is to assume the following *axiom*

$$\mathtt{ofheq} : (a, a' : A) \to \mathsf{heq}(A, A, a, a') \to a = a'.$$

Using this axiom, they prove a congruence lemma $\mathtt{hcongr}_n$ for each $n \geq 1$. The idea is that $\mathtt{hcongr}_n$ is the congruence lemma for dependent functions with n arguments.

Incompatibility with univalence. Recall, from [2], the pathover type family.

Definition 2 (Licata & Brunerie (LICS 2015))**.** *Given a type $B : \mathcal{U}$ and a type family $X : B \to \mathcal{U}$, define the type family*

$$\mathsf{pathover}_B : (b, b' : B) \to (b = b') \to X(b) \to X(b') \to \mathcal{U},$$

by path induction. We denote the type $\mathsf{pathover}_B(b, b', e, a, a')$ by $a =^B_{\langle e \rangle} a'$.

The types heq and $\mathsf{pathover}$ are related as follows.

Lemma 3. *For any $a : A$ and $a' : A'$ we have*

$$\mathsf{heq}(A, A', a, a') \simeq \sum_{e : A = A'} \mathsf{pathover}_{\mathsf{Id} : \mathcal{U} \to \mathcal{U}}(A, A', e, a, a').$$

From this, it follows that the axiom $\mathtt{ofheq}$ implies UIP (uniqueness of identity proofs), and thus, that it is inconsistent with univalence.

Congruence using pathover. In order to synthesize congruence lemmas for type families with arbitrarily many parameters, we must define a pathover type for each such family.

It is conceptually clearer to describe this generalization, and the congruence lemmas, in terms of the category of contexts of our type theory. Given a context Γ and inhabitants $a, b : \Gamma$, one can define an equality context $a = b$ by context induction and path induction ([1, Proposition 3.3.1]). Similarly, given a context extension $\Gamma.\Gamma'$, inhabitants $a, b : \Gamma$, $a' : \Gamma'(a)$, $b' : \Gamma'(b)$, and an equality $e : a = b$, one can define a context of pathovers $a' =_{\langle e \rangle} b'$.

Now, given two context extensions $\Gamma.\Gamma'$ and $\Delta.\Delta'$ and a map $f.f' : \Gamma.\Gamma' \to \Delta.\Delta'$ between them, we can use path induction to prove the following congruence lemma

$$a, b : \Gamma, a' : \Gamma'(a), b' : \Gamma'(b), e_1 : a = b, e_2 : a' =_{\langle e_1 \rangle} b' \vdash$$
$$\mathsf{congr}_{f'}(a, b, a', b', e_1, e_2) : f'(a, a') =_{\langle \mathsf{congr}_f(a,b,e_1) \rangle} f'(b, b').$$

Notice how the type of the congruence lemma for f' uses the congruence lemma for f.

Main result 4. *We give an algorithm to automatically state and prove congruence lemmas for any dependent function.*

The main complication is in correctly characterizing the identity types of contexts. This becomes apparent in the following example.

Example 5. The congruence lemma for $\mathsf{cons} : (n : \mathbb{N}) \to A \to \mathsf{vec}_A(n) \to \mathsf{vec}_A(\mathsf{succ}(n))$ is

$$\mathsf{congr}_{\mathsf{cons}}(n, m, x, y, xs, ys, e_1, e_2, e_3) : \mathsf{cons}(n, x, xs) =_{\langle \mathsf{congr}_{\mathsf{succ}}(e_1) \rangle} \mathsf{cons}(m, y, ys)$$

where $e_1 : n = m$, $e_2 : x = y$, $e_3 : xs =_{\langle e_1 \rangle} ys$, and $\mathsf{congr}_{\mathsf{succ}} : (n, m : \mathbb{N}) \to (n = m) \to \mathsf{succ}(n) = \mathsf{succ}(m)$. We see that, although cons takes as input $x : A$, the type of its codomain does not depend on x, and thus the pathover returned by its congruence lemma should not live over the path $e_2 : x = y$.

This means that we need a representation of contexts that takes dependency into account. We represent contexts as **inverse diagrams**. This is the final ingredient in the procedure. Since the description of the full procedure requires some setting up, we illustrate how it works with an example.

Example 6. We first identify the domain and codomain contexts of cons, and represent them as inverse diagrams

$$\begin{tikzcd} & \mathsf{vec}_A \arrow[d] & & \mathsf{vec}_A \arrow[d] \\ A & \mathbb{N}, & & \mathbb{N} \end{tikzcd}$$

Analyzing the type of cons, we see that cons lives over the context morphism succ

$$\begin{tikzcd} (n : \mathbb{N}).(x : A, xs : \mathsf{vec}_A(n)) \arrow[r, "\mathsf{succ}.\mathsf{cons}"] \arrow[d, two heads] & (n : \mathbb{N}).(xs : \mathsf{vec}_A(n)) \arrow[d, two heads] \\ (n : \mathbb{N}) \arrow[r, "\mathsf{succ}"] & (n : \mathbb{N}). \end{tikzcd}$$

So, inductively, we produce the congruence lemma for succ

$$\mathsf{congr}_{\mathsf{succ}} : (n, m : \mathbb{N}) \to (n = m) \to \mathsf{succ}(n) = \mathsf{succ}(m).$$

Finally, we use path induction, and induction on the inverse diagrams, to correctly characterize the identity types of the domain and codomain of cons. Giving, for example,

$$(e_1 : n = m, e_2 : x = y, e_3 : xs =_{\langle e_1 \rangle} ys)$$

for the domain. Putting these things together, we get the congruence lemma of Example 5.

References

[1] Richard Garner. “Two-dimensional models of type theory”. In: *Mathematical Structures in Computer Science* 19.4 (2009), 687–736. DOI: 10.1017/S0960129509007646.

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

A Model of the Blockchain using Induction-Recursion

Anton Setzer

Dept. of Computer Science, Swansea University, Bay Campus, Swansea SA1 8EN, UK
`a.g.setzer@swansea.ac.uk`

Abstract

We present a model of the blockchain as defined for instance in Bitcoin which is based on transaction dags, and formalise it in Agda. Such a model can nicely be formulated by using (small) induction-recursion, namely by defining inductively the set of transaction dags while recursively defining the set of unspent transaction outputs. We postulate cryptographic functions and their properties and use them to define Merkle trees. In order to guarantee that signatures for claiming transaction outputs cannot be reused, uniqueness of transaction ids is required. If one adds the block number to the coinbase transactions, one can prove this property. It turns out that in the initial implementation of Bitcoin uniqueness of transaction ids was not guaranteed. This was later fixed by adding the block number to coinbase transactions.

Cryptocurrencies are a new approach towards money, where money is stored entirely digitally on the blockchain. The blockchain is a distributed database together with a consensus protocol which decides which instances of the database are the correct ones.

Since cryptocurrencies form a distributed database, they can be used as well to store other information which needs to be uniquely shared world-wide. One main application is the tracing of goods, guaranteeing that goods originate from authentic producers.

There are two main approaches to store the amount stored in a cryptocurrency. One approach is the ledger based approach, where for each user, usually represented by a public/private key pair, the amount of money attributed to him or her is recorded over time. The second approach is the transaction based approach. In that approach the amount a user has is given by the unspent transaction outputs of previous transactions, attributed to that user. The ledger based approach is mathematically much simpler, it can be represented by a function mapping time and public keys to the amount owned by the user at a given time. This is the model used by the cryptocurrency Ethereum. The transaction based model is used in Bitcoin. This model is interesting when considering the application to tracing of goods: The path, a product takes in the transportation chain, can be considered as a sequence of transactions, in which sometimes goods from several sources are combined, or split up between different vendors.

In this talk we present a model of the blockchain based on the transaction model which we have developed in the dependently typed programming language and interactive theorem prover Agda. In this model, the sequence of transactions forms a directed acyclic graph (dag). The transaction dag is defined by small induction-recursion: One defines inductively the transaction dag and transactions while simultaneously recursively defining the unspent transaction outputs.

More precisely one defines inductively the set TXDag of transaction dags and (TX *dag*) of transactions depending on a transaction *dag*:

```
data TXDag : Set where
    genesisDag : TXDag
    txdag      : (dag : TXDag) → TX dag → TXDag
data TX (dag : TXDag) : Set where
    normalTX   : TxInputs dag → List TXOutputfield → TX dag
    coinbase   : Time → List TXOutputfield → TX dag
```

TX consists of normal transactions, given by a list of transaction inputs and a list of transaction outputs, and of coinbase transactions, where miners can transfer to themselves the miner's reward. Transactions dags are formed from the constructors genesisDag, which forms the root of the dag, and txdag, which adds a transaction to a transaction dag. Elements of TXOutputfield are pairs consisting of a public key and the amount given to that public key. In order to define TxInputs, one defines first recursively the unspent transaction outputs (utxo *dag*) of transaction dags:

$$\begin{aligned}&\mathsf{utxo} : \mathsf{TXDag} \to \mathsf{List}\ \mathsf{TXOutput}\\&\mathsf{utxo}\ (\mathsf{txdag}\ dag\ tx) = \mathsf{utxoMinusNewInputs}\ dag\ tx +\!\!+ \mathsf{tx2TXOutputs}\ dag\ tx\end{aligned}$$

When adding a transaction, utxo is obtained by deleting transactions inputs used in the transaction *tx* from the previous dag and adding one new unspent transaction output for each output of the transaction. TxInputs is now given as disjoint lists of unspent transaction outputs.

In the Bitcoin protocol, the transaction dags are encoded as Merkle trees. In the Merkle tree one defines a transaction id for each transaction occurring in a transaction dag. The transaction id is formed by taking the ids for the transaction inputs and the transaction outputs, and hashing the combination of this data. The id of a transaction input is a pair consisting of the transaction id of the transaction and the number of the output. In the formalisation in Agda we postulate the cryptographic functions needed, and postulate properties such as injectivity of the hashing function. Note that, if one took the correct hashing function, it would not be injective. However, since we only postulated the hashing function, injectivity can be fulfilled by instantiating the hashing function with the identity, therefore this doesn't result in an inconsistency.

Furthermore, we postulate the notion of signed messages signed by private keys corresponding to given public keys. Signed transactions are transaction together with a signature for each transaction input (corresponding to the public key for that input). Signed transaction dags are transaction dags consisting of signed transactions.

One can now show that, provided the time values used in coinbase transactions are different, all transaction ids of transactions in a transaction dag are different. Because of this all messages to be signed are different, which shows that signatures from previous transactions cannot be reused for future transactions, ensuring that signatures need to be given by the owners of the public keys in question.

Uniqueness of transaction ids relies on uniqueness of coinbase transactions as given by the additional time field. This field was initially not part of the Bitcoin protocol, and on the Bitcoin blockchain there exist two pairs of transactions with identical transaction ids. Therefore the original protocol allowed the reuse of signatures. This was later fixed by adding the time field to the Bitcoin protocol.

We are currently working on expanding this model by smart contracts, and to use this model to verify the correctness of smart contracts, an area where mistakes have led in the past to substantial financial losses.

References

[1] A. Setzer. Modelling Bitcoin in Agda. *arXiv*, arXiv:1804.06398:27, 17 April 2018. `https://arxiv.org/abs/1804.06398`.

XTT: Cubical Syntax for Extensional Equality (without equality reflection)

Jonathan Sterling[1], Carlo Angiuli[2], and Daniel Gratzer[3]

[1] Carnegie Mellon University
jmsterli@cs.cmu.edu
[2] Carnegie Mellon University
cangiuli@cs.cmu.edu
[3] Aarhus University
gratzer@cs.au.dk

We contribute XTT [14], a cubical reconstruction of Observational Type Theory [2, 3] supporting extensional equality without equality reflection. Following cubical type theory [9, 4, 6], XTT easily obtains function extensionality by defining the equality type in terms of maps out of an abstract interval. Unlike previous cubical type theories, XTT supports *judgmental* unicity of identity proofs (for all $P, Q : \mathsf{Eq}_{i.A}(M, N)$, $P = Q$ judgmentally), and therefore admits substantially simplified Kan operations. Finally, XTT is closed under a cumulative hierarchy[1] of closed universes à la Russell. We hope to integrate XTT into the redtt cubical proof assistant [5] as an implementation of exact equality in the style of two-level type theory [6, 15, 1, 7].

Cubical exact equality The XTT formalism decomposes constructs from OTT into more modular, *judgmental* principles. For instance, rather than defining equality separately at every type and entangling the connectives, we define equality once and for all using interval variables i (below); likewise, rather than ensuring that equality proofs are unique through brute force, we obtain uniqueness of identity proofs indirectly through a judgmental *boundary separation* rule, inspired by Coquand's definition of "Bishop sets" in models of cubical type theory [11]:

EQUALITY INTRODUCTION

$$\frac{\Psi, i \mid \Gamma \vdash M : A \qquad \Psi, i, i=0 \mid \Gamma \vdash M = N_0 : A \qquad \Psi, i, i=1 \mid \Gamma \vdash M = N_1 : A}{\Psi \mid \Gamma \vdash \lambda i.M : \mathsf{Eq}_{i.A}(N_0, N_1)}$$

BOUNDARY SEPARATION

$$\frac{\Psi \mid r : \mathbb{I} \qquad \Psi, r=0 \mid \Gamma \vdash M = N : A \qquad \Psi, r=1 \mid \Gamma \vdash M = N : A}{\Psi \mid \Gamma \vdash M = N : A}$$

In contrast to (univalent) path structure, our cubical account of exact equality satisfies *regularity*: transport in constant type families is judgmentally equal to the identity function. As a result, the cubical equality types of XTT can be used to encode Martin-Löf's identity type, satisfying its β-rule judgmentally.

Model theory, gluing, and canonicity By first developing the model theory of XTT in an algebraic way, we then prove a canonicity theorem for the *initial* model of XTT: any closed term of boolean type is equal to either true or false. We obtain this result using a novel extension of the categorical gluing technique described by Coquand and Shulman [10, 12], by gluing the fundamental fibration of a category of *augmented Cartesian cubical sets* along a cubical nerve. Canonicity expresses a form of "computational adequacy"—in essence, that the equational theory of XTT suffices to derive any equation which ought to hold by (closed) computation—and is one of many syntactical considerations that experience has shown to be correlated to usability. We conjecture that our methods will extend to open terms, allowing us to establish normalization and decidability of the typing relation.

[1] As in previous work [13], we employ an *algebraic* version of cumulativity which does not require subtyping.

References

[1] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. "Extending Homotopy Type Theory with Strict Equality". In: *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. Vol. 62. Dagstuhl, Germany, 2016. DOI: 10.4230/LIPIcs.CSL.2016.21.

[2] Thorsten Altenkirch and Conor McBride. *Towards Observational Type Theory*. 2006. URL: www.strictlypositive.org/ott.pdf.

[3] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational Equality, Now!" In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. Freiburg, Germany, 2007.

[4] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. "Syntax and Models of Cartesian Cubical Type Theory". Feb. 2019. URL: https://github.com/dlicata335/cart-cube.

[5] Carlo Angiuli, Evan Cavallo, Kuen-Bang Hou (Favonia), Robert Harper, Anders Mörtberg, and Jonathan Sterling. *redtt: implementing Cartesian cubical type theory*. URL: http://www.jonmsterling.com/pdfs/dagstuhl.pdf.

[6] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. "Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities". In: *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Vol. 119. Dagstuhl, Germany, 2018. DOI: 10.4230/LIPIcs.CSL.2018.6.

[7] Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. *Two-Level Type Theory and Applications*. 2017. arXiv: 1705.03307.

[8] James Chapman, Fredrik Nordvall Forsberg, and Conor McBride. "The Box of Delights (Cubical Observational Type Theory)". Jan. 2018. URL: https://github.com/msp-strath/platypus/blob/master/January18/doc/CubicalOTT/CubicalOTT.pdf.

[9] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. "Cubical Type Theory: a constructive interpretation of the univalence axiom". In: *IfCoLog Journal of Logics and their Applications* 4.10 (Nov. 2017). URL: http://www.collegepublications.co.uk/journals/ifcolog/?00019.

[10] Thierry Coquand. *Canonicity and normalization for Dependent Type Theory*. Oct. 2018. arXiv: 1810.09367.

[11] Thierry Coquand. *Universe of Bishop sets*. Feb. 2017. URL: http://www.cse.chalmers.se/~coquand/bishop.pdf.

[12] Michael Shulman. "Univalence for inverse diagrams and homotopy canonicity". In: *Mathematical Structures in Computer Science* 25.5 (2015). DOI: 10.1017/S0960129514000565.

[13] Jonathan Sterling. *Algebraic Type Theory and Universe Hierarchies*. Dec. 2018. arXiv: 1902.08848 [math.LO].

[14] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. "Cubical Syntax for Reflection-Free Extensional Equality". In: *Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Vol. 131. 2019. DOI: 10.4230/LIPIcs.FSCD.2019.32. arXiv: 1904.08562.

[15] Vladimir Voevodsky. "A simple type system with two identity types". Feb. 2013. URL: https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf.

Proof Irrelevance and Predicate Subtyping in Dedukti

François Thiré[1][2] and Gaspard Férey[1][2][3]

[1] INRIA
[2] LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay
[3] MINES ParisTech, PSL University

The logical framework Dedukti is an extension of LF relying on user-defined rewrite rules. It has met some success in the encoding of several logics such as First-Order Logic, Higher-Order Logic, Pure Type Systems or The Calculus of Inductive Constructions [1] and is now effectively used to translate proofs from one system to another as done in [5]. Gilbert [3] showed that the core of the PVS system is an extension of Higher-Order Logic with predicate subtyping.

Predicate subtyping is a type theoretic construction restricting a type to its elements such that a certain property is provable.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash P\ t \qquad \Gamma \vdash \{x : A \mid P\ x\} : \mathit{Type}}{\Gamma \vdash t : \{x : A \mid P\ x\}}\ \textsc{SubTypingIntro}$$

This feature allows, for example, to express the type of even numbers, $\{x : \mathbb{N} \mid \text{Even } x\}$, from the predicate Even on $\mathbb{N}$ which holds whenever its argument is an even number. This feature can be obtained or encoded in other logics by means of dependent pairs, the subtype of even numbers becoming then the type of natural numbers together with a proof of their parity. However, in that case, for any two different proofs p and q that 2 is even, one will derive two different terms $\langle 2, p\rangle$ and $\langle 2, q\rangle$ of type $\{x : \mathbb{N} \mid \text{Even } x\}$ even though they correspond to the same PVS expression. Indeed predicate subtyping as implemented in PVS is a *proof irrelevant* construction in the sense that the proof of the predicate is omitted in the introduction rule.

In this work we define a slight extension of Dedukti allowing to encode a form of judgmental proof irrelevance i.e. convertibility of all proofs of the same theorem. To do so, we allow the annotation of some symbols as *private*, meaning that these symbols may only occur in the conversion without ever occuring in the translation of a theorem or its proof. This idea of *private* symbols goes beyond the encoding of predicate subtyping into Dedukti and can be used to encode a proof irrelevant theory or proof irrelevant features of a system such as the recent universe SPROP in Coq [4].

Private Definitions A natural way to encode proof irrelevance is to ensure that any proof of a statement, if there is any, rewrites to the same canonical proof. This is easily done by declaring a symbol `hilbert` mapping propositions A to a proof of A such that `hilbert A` is the canonical proof of the statement A. This naturally breaks the consistency of the logic as it allows to build a proof of any statement, including `False`. To overcome this issue, we restrict the use of `hilbert`. For that purpose, we introduce an extension of Dedukti allowing the declaration of *private* symbols. This can be done via the keyword `private` which has the following semantics: A *private* symbol may never occur in the translation of a theorem or of its proof. However, it may be used in the rewrite rules defining the encoding. This way, we guarantee statically that proofs never contain private symbols, thus forbidding unlawful proofs of `False`.

Proof irrelevance A variant of Pure Type Systems with a proof irrelevant sort as in [2] can be encoded using the aforementioned `hilbert` *private* proof constructor together with a

```
A : Type.   a : A.   b : A.
private pr : A -> Type.
pu         : A -> Type.
[] pu a --> pr a.        (; Fine ;)
[] pr a --> pr b.        (; Fine ;)
```

```
(; Proofs generated by the translation ;)

my_axiom   : pr a.                    (; Error ;)
def my_thm                := pr a.   (; Error ;)
def my_prf : proof my_thm := pr a.   (; Error ;)
```

make_proof constructor. The latter collapses distinct proofs of a statement A into the canonical proof built with the hilbert symbol. However, this requires all translated proofs to use the make_proof symbol. Such construction also provide a way to encode the SPROP universe as described in [4].

```
Prop : type.
iproof : Prop -> Type.   proof : Prop -> Type.
private hilbert : A : Prop -> iproof A.
def make_proof  : A : Prop -> proof A -> iproof A.
[A,prf] make_proof A prf --> hilbert A.
```

```
A : Prop. B : Prop. AorB : Prop.
left  : iproof A -> proof AorB.
right : iproof B -> proof AorB.
(; make_proof AorB (left  a) == ;)
(; make_proof AorB (right b)  ;)
```

Predicate subtyping Faithfully encoding PVS predicate subtyping requires a finer notion of proof irrelevance. Indeed only the construction of a subtype predicate may need to be proof irrelevant while the previous method contaminates all propositions with proof irrelevance. This can also be a problem when translating PVS libraries to proof relevant systems. To put this plague into quarantine, we discharge proof irrelevance to the constructor of a subtype predicate.

```
Sig : Nat -> (Nat -> Prop) -> Type.
def     mk_sig  : (n : Nat) -> (P : (Nat -> Prop)) -> p : proof (P n) -> Sig n P.
private mk_sig' : (n : Nat) -> (P : (Nat -> Prop)) -> Sig n P.
[n, P, prf] mk_sig n P prf --> mk_sig' n P. (; Irrelevance of prf ;)

even : Nat -> Prop.   p : proof (even 2).   q : proof (even 2).
def prf_p : Sig 2 even := mk_sig 2 even p.  (;      rewrites to  mk_sig' 2 even ;)
def prf_q : Sig 2 even := mk_sig 2 even p.  (; also rewrites to  mk_sig' 2 even ;)
```

In the above example, both translated definitions prf_p and prf_q are rewritten to a same normal form independant from the provided proof witness. While using make_proof would enforce all the proofs of even 2 to be proof irrelevant, with this method only the third argument of the constructor mk_sig is. Using this proof irrelevant constructor mk_sig, we have taken down the last bastion preventing the encoding of PVS into Dedukti.

References

[1] Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the $\lambda\Pi$-calculus modulo theory and in the Dedukti system. In *TYPES: Types for Proofs and Programs*, Novi SAd, Serbia, May 2016.

[2] Lukasz Czajka. A Shallow Embedding of Pure Type Systems into First-Order Logic. Dagstuhl, Germany, 2018.

[3] Frédéric Gilbert. *Extending higher-order logic with predicate subtyping : Application to PVS*. PhD thesis, Sorbonne Paris Cité, France, 2018.

[4] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *PACMPL*, 3(POPL):3:1–3:28, 2019.

[5] François Thiré. Sharing a library between proof assistants: Reaching out to the HOL family. In *LFMTP@FSCD*, volume 274 of *EPTCS*, pages 57–71, 2018.

A General Framework for the Semantics of Type Theory

Taichi Uemura

University of Amsterdam, Amsterdam, the Netherlands
t.uemura@uva.nl

Dybjer [4] introduced *categories with families* as a notion of a model of basic dependent type theory. Extending categories with families, one can define notions of models of dependent type theories such as Martin-Löf type theory [5], two-level type theory [1] and cubical type theory [3]. The way to define a model of a dependent type theory is by adding algebraic operations corresponding to type and term constructors, and it is a kind of routine. However, as far as the author knows, there are no general notions of a "type theory" and a "model of a type theory" that include all of these examples. In this talk, we propose general notions of a type theory and a model of a type theory to unify semantics of type theories based on categories with families.

Awodey [2] pointed out that a category with families is the same thing as a *representable map* of presheaves and that type and term constructors are modeled by algebraic operations on presheaves. Inspired by this work, we introduce a logical framework in which one can declare a family of *representable types* as well as a family of ordinary types. More precisely, our logical framework has a sort $\square$ of types and a subsort $* \subset \square$ of representable types. It also has extensional identity types in $\square$ and dependent function types of the form

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : \square}{\Gamma \vdash (x : A) \to B : \square}.$$

Then, for instance, basic dependent type theory can be encoded to the signature consisting of the following constants.

$$\vdash \mathsf{Type} : \square$$
$$A : \mathsf{Type} \vdash \mathsf{el}(A) : *$$

This signature tells us what a model of basic dependent type theory is: it consists of a category $\mathcal{S}$ of contexts, a presheaf U over $\mathcal{S}$, which corresponds to Type, and a representable map $E \to U$ of presheaves over $\mathcal{S}$, which corresponds to el.

One can encode type constructors in a natural way. For instance, Π-types are encoded by constants

$$A : \mathsf{Type}, B : \mathsf{el}(A) \to \mathsf{Type} \vdash \Pi(A, B) : \mathsf{Type}$$
$$A : \mathsf{Type}, B : \mathsf{el}(A) \to \mathsf{Type}, b : (x : \mathsf{el}(A)) \to \mathsf{el}(Bx) \vdash \mathsf{abs}(b) : \mathsf{el}(\Pi(A, B))$$
$$A : \mathsf{Type}, B : \mathsf{el}(A) \to \mathsf{Type}, f : \mathsf{el}(\Pi(A, B)), a : \mathsf{el}(A) \vdash \mathsf{app}(A, B, f, a) : \mathsf{el}(Ba)$$

and some equations.

We can encode more complicated type theory. *Cubical type theory* [3] is a dependent type theory with an interval, cofibrant propositions and composition operations. We encode the interval to a representable type $\vdash \mathbb{I} : *$ equipped with some constants including end-points $0, 1 : \mathbb{I}$. Cofibrant propositions are encoded by types $\vdash \mathsf{Cof} : \square$ and $P : \mathsf{Cof} \vdash \mathsf{true}(P) : *$ such that $\mathsf{true}(P)$ is a proposition in the sense that any two elements of $\mathsf{true}(P)$ are equal. The composition operation is encoded to a constant

$$\begin{aligned} &A : \mathbb{I} \to \mathsf{Type}, P : \mathsf{Cof}, a : \mathsf{true}(P) \to (i : \mathbb{I}) \to \mathsf{el}(Ai), \\ &\qquad a_0 : \mathsf{el}(A0), e : (x : \mathsf{true}(P)) \to ax0 = a_0 \vdash \mathsf{comp}(A, P, a, a_0, e) : \mathsf{el}(A1) \end{aligned}$$

and an equation $ax1 = \mathsf{comp}(A, P, a, a_0, e)$ for $x : \mathsf{true}(P)$.

With this logical framework, we establish basic results in the semantics of type theory. For a signature Σ of our logical framework, we define a notion of a *theory over* Σ and establish a correspondence between theories over Σ and models of Σ: for each theory K over Σ, we construct the *syntactic model of* Σ *generated by* K; for each model $\mathcal{S}$ of Σ, we construct the *internal language of* $\mathcal{S}$. Categorically, these constructions yields a bi-adjunction between the (locally discrete) 2-category of theories over Σ and the 2-category of models of Σ. Moreover, this bi-adjunction induces a bi-equivalence between the 2-category of theories over Σ and a full sub-2-category of the 2-category of models of Σ.

References

[1] Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. *Two-Level Type Theory and Applications.* 2017. arXiv: 1705.03307v2.

[2] Steve Awodey. "Natural models of homotopy type theory". In: *Mathematical Structures in Computer Science* 28.2 (2018), pp. 241–286. DOI: 10.1017/S0960129516000268.

[3] Cyril Cohen et al. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015).* Ed. by Tarmo Uustalu. Vol. 69. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 5:1–5:34. DOI: 10.4230/LIPIcs.TYPES.2015.5.

[4] Peter Dybjer. "Internal Type Theory". In: *Types for Proofs and Programs: International Workshop, TYPES '95 Torino, Italy, June 5–8, 1995 Selected Papers.* Ed. by Stefano Berardi and Mario Coppo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 120–134. DOI: 10.1007/3-540-61780-9_66.

[5] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Studies in Logic and the Foundations of Mathematics* 80 (1975), pp. 73–118. DOI: 10.1016/S0049-237X(08)71945-1.

Guarded Recursion in Agda via Sized Types

Niccolò Veltri[1] and Niels van der Weide[2]

[1] Department of Computer Science, IT University of Copenhagen, Denmark
nive@itu.dk
[2] Institute for Computation and Information Sciences, Radboud University, Nijmegen, The Netherlands
nweide@cs.ru.nl

Sized types and guarded recursion are two techniques used to ensure the productivity of recursively defined elements of coinductive types. Productivity means that each finite part of the output only depends on a finite part of the input, and it is necessary to ensure consistency of the type system. Programming with coinductive types becomes less convoluted when employing these techniques instead of traditional approaches relying on strict syntactic checks. In this work, we show how guarded recursion can be simulated in Agda using sized types.

Sized types [5] are types annotated with an abstract ordinal indicating the number of possible unfoldings that can be performed on elements of the type. Sized types are implemented in Agda and can be used in combination with coinductive records [1] to specify coinductive types.

Guarded recursion [7] is a different approach where the type system is enhanced with a modality, called "later" and written $\triangleright$, encoding time delay in types. The later modality comes with a general fixpoint combinator for programming with productive recursive functions and allows the specification of guarded recursive types. These types can be used in combination with clock quantification to define coinductive types [2].

To simulate guarded recursion via sized types, we formalize the syntax of a simple type theory for guarded recursion in Agda, which we call **GTT**. Then we prove the syntax sound w.r.t. a categorical semantics specified in terms of sized types[1].

The Syntax of GTT

The language **GTT** is a variant of Atkey and McBride's type system for productive coprogramming [2]. In Atkey and McBrides calculus, all judgments are indexed by a clock context, which may contain several different clocks. They extend simply typed lambda calculus with two additional type formers: a modality $\triangleright$ for encoding time delay into types and universal quantification over clock variables $\forall$, which is used in combination with $\triangleright$ to specify coinductive types.

The judgements of **GTT** depend on a clock context which can only be empty or contain a single clock. The types of **GTT** include the $\triangleright$ modality and a modality $\square$, which is a nameless analogue of Atkey and McBride's universal clock quantification. The $\square$ type former maps a type in the singleton clock context to one in the empty clock context. Guarded recursive types are defined using a least fixpoint type former μ. We also have a weakening operation on types, which maps a type in the empty clock context to one in the singleton clock context, and a similar operations on contexts. Clouston *et al.*[4] also studied a guarded variant of lambda calculus extended with a $\square$ operation, which they call "constant". **GTT** differs from their calculus in that our judgments are indexed by a clock context and it has the benefit of allowing a more appealing introduction rule for the $\square$ modality.

We only allow clock contexts in **GTT** to contain at most one clock variable, because Agda's support for sized types is tailored to types depending on exactly one size, or on a finite but

[1]The full Agda formalization can be found at https://github.com/niccoloveltri/agda-gtt.

precise number of sizes, which makes it cumbersome to work with types depending on clock contexts containing an indefinite number of clocks.

The terms of **GTT** include operations box and unbox, which are the introduction and elimination rules for the $\square$ modality. In Atkey and McBride's system, these rules correspond to clock quantification and clock application respectively. In the rule for clock quantification, they add a side condition requiring the universally quantified clock to not appear free in the variable context. In **GTT**, we achieve this by requiring box to be applicable only to terms over a weakened context.

GTT also has a delayed fixpoint combinator dfix. This term takes as input a productive recursive definition, represented by a function of type $\triangleright A \rightarrow A$, and returns an element of $\triangleright A$. Using dfix we can define the usual fixpoint operator, returning a term of type A instead of $\triangleright A$.

Categorical Semantics of GTT

For the denotational semantics, we use a variation of the topos of trees [3]. Instead of natural numbers, we take the preorder of sizes as the indexing category of the presheaves. Types and contexts of **GTT** in the empty clock contexts are interpreted as sets, while types and contexts in the singleton clock contexts are interpreted as antitone sized types. The simple type and term formers are interpreted using the standard Kripke semantics.

Following Møgelberg's interpretation of universal clock quantification [6], we model the $\square$ modality by taking limits. For the semantic later modality, we adapt the definition in the topos of trees to our setting. Given a presheaf A, the action of $\blacktriangleright A$ on a size i is given by taking the limit of A on all sizes strictly smaller than i. The semantic fixpoint operator is defined using self-application. The productivity of this construction relies on sizes being a well-ordered set. This gives rise to a consistent interpretation of **GTT**.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming Infinite Structures by Observations. In *POPL*, pages 27–38, 2013.

[2] Robert Atkey and Conor McBride. Productive Coprogramming with Guarded Recursion. In *ICFP*, pages 197–208, 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. *Logical Methods in Computer Science*, 8(4), 2012.

[4] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *FoSSaCS*, pages 407–421, 2015.

[5] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL*, pages 410–423, 1996.

[6] Rasmus Ejlers Møgelberg. A Type Theory for Productive Coprogramming via Guarded Recursion. In *CSL-LICS*, pages 71:1–71:10, 2014.

[7] Hiroshi Nakano. A Modality for Recursion. In *LICS*, pages 255–266, 2000.

Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types

Andrea Vezzosi[1], Anders Mörtberg[2][3], and Andreas Abel[4]

[1] IT University Copenhagen, Copenhagen, Denmark
[2] Stockholm University, Stockholm, Sweden
[3] Carnegie Mellon University, Pittsburgh, USA
[4] Chalmers and Gothenburg University, Gothenburg, Sweden

A core idea in programming and mathematics is *abstraction*: the exact details of how an object is represented should not affect its abstract properties. The principle of univalence captures this by extending the equality on the universe of types to incorporate equivalent types. This provides a form of abstraction, or invariance up to equivalence, in the sense that equivalent types will share the same structures and properties. The fact that equality is *proof relevant* in dependent type theory is the key to enabling this. The data of an equality proof can store the equivalence and transporting along this equality should then apply the function underlying the equivalence. In particular, this allows programs and properties to be transported between equivalent types, hereby increasing modularity and decreasing code duplication. A concrete example are the equivalent representations of natural numbers in unary and binary format. In a univalent system it is possible develop theory about natural numbers using the unary representation, but compute using the binary representation, and as the two representations are equivalent they share the same properties.

The principle of univalence is the major new addition in Homotopy Type Theory and Univalent Foundations (HoTT/UF) [The Univalent Foundations Program, 2013]. However, these new type theoretic foundations add univalence as an *axiom* which disrupts the good constructive properties of type theory. In particular, if we transport addition on binary numbers to the unary representation we will not be able to compute with it as the system would not know how to reduce the univalence axiom. Cubical Type Theory (CTT) [Cohen et al., 2015a] addresses this problem by introducing a novel representation of equality proofs and thereby providing computational content to univalence. This makes it possible to constructively transport programs and proofs between equivalent types. This representation of equality proofs has many other useful consequences, in particular functional and propositional extensionality and the equivalence between bisimilarity and equality for coinductive types [Vezzosi, 2017].

Dependently typed functional languages such as `Agda`, `Coq`, `Idris`, and `Lean`, provide rich and expressive environments supporting both programming and proving within the same language. However, the extensionality principles mentioned above are not available out of the box and need to be assumed as axioms just as in HoTT/UF. Unsurprisingly, this suffers from the same drawbacks as it compromises the computational behavior of programs that use these axioms. It even makes subsequent proofs more complicated as equational properties do not hold by computation.

So far, CTT has been developed with the help of a prototype `Haskell` implementation called `cubicaltt` [Cohen et al., 2015b], but it has not been integrated into one of the main dependently typed functional languages. Recently, an effort was made, using `Coq`, to obtain effective transport for restricted uses of the univalence axiom [Tabareau et al., 2018], because, as the authors mention, *"it is not yet clear how to extend [proof assistants] to handle univalence internally"*.

We achieve this, and more, by making `Agda` into a cubical programming language with native support for univalence and higher inductives types (HITs). We call this extension `Cubical Agda` [2019] as it incorporates and extends CTT. In addition to providing a fully constructive univalence theorem, `Cubical Agda` extends the theory by allowing proofs of equality by copatterns, HITs as in Coquand et al. [2018] with nested pattern matching, and interval and partial pre-types. The extension of dependent (co)pattern matching [Cockx and Abel, 2018] to the equality type allows for convenient programming with HITs and univalence. We demonstrate this by the proof that the torus is equal to two circles in `Cubical Agda`.

```
data S¹ : Set where          data Torus : Set where
  base : S¹                    point  : Torus
  loop : base ≡ base           line1  : point ≡ point
                               line2  : point ≡ point
                               square : PathP (λ i → line1 i ≡ line1 i) line2 line2

t2c : Torus → S¹ × S¹                   c2t : S¹ × S¹ → Torus
t2c point        = (base   , base)      c2t (base , base)     = point
t2c (line1 i)    = (loop i , base)      c2t (loop i , base)   = line1 i
t2c (line2 j)    = (base   , loop j)    c2t (base , loop j)   = line2 j
t2c (square i j) = (loop i , loop j)    c2t (loop i , loop j) = square i j
```

The proof that t2c and c2t are inverses is just reflexivity in each of the four cases. We can then turn the isomorphism into an equality, using the implementation of univalence.

While this is a rather elementary result in topology it had a surprisingly non-trivial proof in HoTT because of the lack of definitional computation for higher constructors [Sojakova, 2016, Licata and Brunerie, 2015]. With the additional definitional computation rules and pattern-matching of `Cubical Agda` this proof is now almost entirely trivial.

References

Agda developers. *Agda 2.6.0 documentation*, 2019. http://agda.readthedocs.io/en/v2.6.0/.

J. Cockx and A. Abel. Elaborating dependent (co)pattern matching. *Proc. ACM Program. Lang.*, 2 (ICFP), 2018. http://doi.acm.org/10.1145/3236770.

C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *TYPES'15*, vol. 69 of *LIPIcs*. Dagstuhl, 2015a. https://doi.org/10.4230/LIPIcs.TYPES.2015.5.

C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubicaltt. https://github.com/mortberg/cubicaltt, 2015b.

T. Coquand, S. Huber, and A. Mörtberg. On higher inductive types in cubical type theory. In *LICS'18*. ACM, 2018. https://doi.org/10.1145/3209108.3209197.

D. R. Licata and G. Brunerie. A cubical approach to synthetic homotopy theory. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'15*. ACM, 2015.

K. Sojakova. The Equivalence of the Torus and the Product of Two Circles in Homotopy Type Theory. *ACM Transactions on Computational Logic*, 17(4), 2016.

N. Tabareau, E. Tanter, and M. Sozeau. Equivalences for free: Univalent parametricity for effective transport. *Proc. ACM Program. Lang.*, 2(ICFP), 2018. http://doi.acm.org/10.1145/3236787.

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

A. Vezzosi. Streams for cubical type theory. http://saizan.github.io/streams-ctt.pdf, 2017.

Author Index