EUTYPES-TYPES 2020 - Abstracts

Ugo de' Liguoro and Stefano Berardi (eds.)

Turin, March 2020

Foreword

This volume contains the abstracts of the talks accepted for presentation at the 26th International Conference on Types for Proofs and Programs, EUTYPES-TYPES 2020, in Turin. But unfortunately, the circumstances have been not favorable; being planned on the dates of 2 - 5 March 2020 in Turin, the workshop has not been held, because of the sanitary emergency caused by the spreading of the SARS-CoV-2 virus infection.

As the title itself suggests, the talks illustrate recent research and ongoing work in the area of type theory, cross cutting the fields of logic, category theory and computer science. The conference was structured in two parts: the first part was dedicated to the Cost Action EUTypes ca 15123, that started on 21 March 2016 in Brussels, and it was planned as the final meeting. The second part was the 26th edition of the TYPES workshop.

The Cost Action EUTypes, led by Herman Geuvers (chair) and Tarmo Uustalu (vice-chair), has been devoted to type theory and its numerous applications in computer science; it has been structured in four working groups:

WG1: Theoretical Foundations

WG2: Type-theoretic tools

WG3: Types for programming

WG4: Types for verification

The WG are referred to in the titles of the first eight sections of this collection.

The remaining six sessions correspond to the program of the third and fourth days of the conference, dealing with topic strictly related to the formers, further exploring the fields of types and computation theory, of logic and λ-calculi, proof-theory, category theory and logic. They include, but are not limited to, the following:

- 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 abstracts testify how reach is the research about types, both in theoretical and in practical directions.

Ugo de' Liguoro and Stefano Berardi

Turin, 1 March 2020

Contents

1 Homotopy and categories in type theory (WG1)

A Semi-simplicial Model of System F in Dependent Type Theory Modulo Rewriting

Bruno Barras and Rehan Malak *

Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria
Laboratoire Spécification et Vérification, 94235, Cachan, France

1 Introduction

The consistency of the Homotopical Type Theory [Pro13] and the Cubical Type Theories is currently based on presheaf models on simplicial or cubical categories: the initial Kan-simplicial set model of HoTT [KLV12] and the various cubical flavors inspired by [CCHM18]

$$\vdash T : Type \quad \rightsquigarrow \quad [\mathcal{C}^{\mathrm{op}}, \mathcal{S}\mathrm{et}] \qquad \text{where } \mathcal{C} = \Delta \text{ or } \square \text{ or } \cdots$$

Intensional Type Theories are just good enough to encode those models as long as we restrict to a fixed level of truncation [BCH15]. However, the formalization of semi-simplicial sets has shown to be notoriously difficult in the general case [Her15] because the definition requires coherence conditions that could not have been solved definitionally so far.

The current solution is to work in a two-level type theory [ACKS19, ACK16, BT12, PL15], where an *external* (or logical) equality is used to express those conditions. There also exists variants [Voe12] where a reflection rule allows to use definitional equality for the coherence conditions. The drawback is that in both cases coherence conditions are expressed with an equality which is not decidable, and may thus require extra work from the user.

We investigate the power of $\lambda\Pi/\mathcal{R}$, a dependently typed λ-calculus extended with user-defined rewrite rules and implemented by the Dedukti proof-assistant [ABC+16, Ded20], to formalize the category of semi-simplicial sets. This is then turned into a model of system F. Rewrite rules allows to express the coherence conditions in the definitional equality, while remaining in a decidable fragment.

The Agda proof-assistant has since developed an experimental feature to extend the definition equality with rewrite rules [Coc19].

2 Semi-simplicial model : induction on the dimension

First we define the base category $\mathcal{C}$. In our work, the type of objects is set to the ordered natural numbers (`nat`), representing the dimension. `F` is the type of morphisms of $\mathcal{C}$.

We use a rewriting rule for the associativity of composition in $\mathcal{C}$. The `&` prefix before a variable means that it is a pattern of a rewriting `rule` associated to `comp`.

```
// Morphisms and (associative) composition
symbol F (i : nat) (j : nat) : TYPE
symbol comp (i j k : nat) : F i j ⇒ F j k ⇒ F i k //!\ order
rule   comp &i &k &l (comp &i &j &k &f &g) &h →
       comp &i &j &l &f  (comp &j &k &l &g &h)
```

*This research has benefited of the financial support from the French Labex Digicosme.

This is only a partial definition of $\mathcal{C}$, showing the generality of the construction to many presheaf models. The semi-simplicial set model is obtained by setting `F i j` to be the injective order-preserving maps from $[0..i]$ to $[0..j]$. They serve as an index for the i-simplices included in a j-simplex.

Semi-simplicial sets are represented by a type of codes `ssset`. The sets associated to each dimension `SX` are parametrized by skeletons as in [Her15]. They have to be defined each time a new code is introduced. The definition requires auxiliary definitions listed below:

```
// A Type of ``codes'' for the semi-simplicial sets
symbol ssset : TYPE
// Skeleton: simplices of dim <n included in the i-simplex
symbol SB (L : ssset) (n : nat) (i : nat) : TYPE
// Type of n-simplices based on skeleton b, to be defined for each code
symbol SX (L : ssset) (n : nat) (b : SB L n n) : TYPE
// Effect of the functor on morphisms of C + coherence condition
symbol mapSB L n i j : F i j ⇒ SB L n j ⇒ SB L n i
rule   mapSB &L &n &i &j &f (mapSB &L &n &j &k &g &b) →
       mapSB &L &n &i &k (comp &i &j &k &f &g) &b
```

We have *assumed* that some data exists and satisfies some coherence condition at all dimensions. Now we can *define* `SB` and `mapSB`[1] by induction on n, and check (using the `assert` command) that it satisfies the above coherence condition. Defining directly constants `SB` and `mapSB` would make this check trivial, so we introduce new symbols for the intended definition:

```
// SBS(L,n) is the intended definition for SB(L,n+1)
symbol SBS (L : ssset) (n : nat) (i : nat) : TYPE
symbol mapSBS (L : ssset) (n i j : nat)  // idem for mapSB
              (f : F i j) (b : SBS L n j) : SBS L n i
// Checking the coherence of the intended def above
assert { ... } mapSBS L n i j f (mapSBS L n j k g b) ≡
                 mapSBS L n i k (comp i j k f g) b
```

Once this has been checked we can make the definition effective:

```
rule   SB &L (nS &n) &i              → SBS &L &n &i
rule   mapSB &L (nS &n) &i &j &f &b → mapSBS &L &n &i &j &f &b
```

Following the above definition scheme by induction on the dimension, we have defined (1) the category of semi-simplicial sets equipped with (2) cartesian product ($\times$) and exponential ($\to$) and (3) a generalized impredicative cartesian product $\forall_{i:I} A(i)$ [2] Using the above definitions, we could build a model of the polymorphic λ-calculus system F [3] , where types are interpreted as semi-simplicial sets.

We are currently formalizing dependent type theories with Categories with Families [Dyb95]. As the models are changing rapidly, one would like to remain as much as possible agnostic to one particular choice of category and fibration morphisms. In all cases, one has to generalize and complete our code with : families of objects, degeneracy maps, Kan condition. We hope that this will reveal the full potential of user-defined rewriting rules. On the meta-theory level, one has also to check that they form a normalizing, confluent rewriting system. For this, we count on the interoperability between Dedukti and third-party specialized tools.[4]

[1] The actual definition of the base and step cases of the definition are omitted due to space constraints.

[2] Since $\lambda\Pi/\mathcal{R}$ is a weak predicative theory, we need to assume the existence of an impredicative universe.

[3] we cannot prove that this model is parametric although it might be if the impredicative universe happens to be parametric

[4] cf the `--confluence` and `--termination` options of Dedukti

References

[ABC+16] 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 λΠ-calculus modulo theory and in the Dedukti system. In *TYPES: Types for Proofs and Programs*, Novi SAd, Serbia, May 2016.

[ACK16] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending homotopy type theory with strict equality. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[ACKS19] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. 2019. arXiv:1705.03307.

[BCH15] Bruno Barras, Thierry Coquand, and Simon Huber. A generalization of the Takeuti–Gandy interpretation. *Mathematical Structures in Computer Science*, 25(5):1071–1099, 2015.

[BT12] Simon Boulier and Nicolas Tabareau. Model structure on the universe in a two level type theory. https://hal.archives-ouvertes.fr/hal-01579822/file/main.pdf, 2012.

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Coc19] Jesper Cockx. Type theory unchained: Extending type theory with user-defined rewrite rules. https://jesper.sikanda.be/files/type-theory-unchained-draft-2019-11-12.pdf, November 2019. Submitted to the TYPES 2019 post-proceedings (PDF).

[Ded20] Deducteam. Lambdapi, proof assistant based on the λΠ-calculus modulo rewriting. https://github.com/Deducteam/lambdapi, 2020.

[Dyb95] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995.

[Her15] Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, 25(Special issue 05):16, June 2015.

[KLV12] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. 2012. arXiv:1211.2851.

[PL15] Fedor Part and Zhaohui Luo. Semi-simplicial types in logic-enriched homotopy type theory. 2015. arXiv:1506.04998.

[Pro13] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, Institute for Advanced Study, 2013.

[Voe12] Vladimir Voevodsky. Homotopy type systems. https://ncatlab.org/homotopytypetheory/show/Homotopy+Type+System, 2012.

Constructing a universe for the setoid model

Thorsten Altenkirch[1]*, Ambrus Kaposi[2], Christian Sattler[1]†, and Filippo Sestini[1]

[1] School of Computer Science, University of Nottingham, UK
{psztxa,pszcs1,psxfs5}@nottingham.ac.uk
[2] Eötvös Loránd University, Budapest, Hungary
akaposi@inf.elte.hu

The setoid interpretation gives rise to a model of a type theory with functional and propositional extensionality. It is thus a way to explain extensionality in a type-theoretic and computationally adequate way [Alt99]. The setoid model has been recently adapted into Setoid Type Theory (SeTT), which is justified by a syntactic translation into a very basic *target theory* [ABKT19]. This translation relies on the existence in the target theory of a universe of definitionally proof-irrelevant propositions Prop, as recently implemented in Coq and Agda [GCST19]. The setoid interpretation models a universe of propositions where equality of propositions is logical equivalence, thus providing a very basic instance of univalence. However, we also would like to be able to equip SeTT with a universe of setoids. This universe must in turn be a setoid, hence in particular it cannot be univalent, but rather its equivalence relation must reflect simple equality of codes.

To provide such a universe we need to define a setoid $\mathcal{U} : \mathsf{Setoid}$ and a family of setoids $\mathsf{El} : \mathcal{U} \to \mathsf{Setoid}$, with codes for basic type formers like Π-types and booleans. This can be obtained as an inductive-recursive type.

$$
\begin{aligned}
&\mathsf{data}\ \mathcal{U} : \mathsf{Set}_1\\
&\_ \sim_{\mathcal{U}} \_ : \mathcal{U} \to \mathcal{U} \to \mathsf{Prop}_1\\
&\mathsf{El} : \mathcal{U} \to \mathsf{Set}\\
&\_ \vdash \_ \sim_{\mathsf{El}} \_ : \{a\ a' : \mathcal{U}\} \to a \sim_{\mathcal{U}} a' \to \mathsf{El}\ a \to \mathsf{El}\ a' \to \mathsf{Prop}
\end{aligned}
$$

However, we don't want to assume inductive-recursive types in the basic type theory which is the target of the setoid translation. We know that we can translate basic instances of induction-recursion into inductive families using the equivalence of I-indexed families of sets and sets over I, i.e. $[I, \mathsf{Set}_i] \simeq \mathsf{Set}_i/I$ for $I : \mathsf{Set}_i$. For example, the inductive-recursive definition of a universe $\mathsf{U} : \mathsf{Set}_1$ and $\mathsf{El} : \mathsf{U} \to \mathsf{Set}$ with Π-types and Booleans is[1]:

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{U}\ : \mathsf{Set}_1\\
&\quad \mathsf{bool} : \mathsf{U}\\
&\quad \mathsf{pi} : (A : \mathsf{U}) \to (\mathsf{El}\ A \to \mathsf{U}) \to \mathsf{U}\\
&\mathsf{El}\ \mathsf{bool} = \mathbb{2}\\
&\mathsf{El}\ (\mathsf{pi}\ A\ B) = (a : \mathsf{El}\ A) \to \mathsf{El}\ (B\ a)
\end{aligned}
$$

We can model this as an inductive type $\mathsf{in\text{-}U}$ that *carves out* all types in Set that are in the

*Supported by USAF grant FA9550-16-1-0029.
†Supported by USAF grant FA9550-16-1-0029.

[1] In Agda, this definition would also go through if U were in Set, but this seems to be a non-conservative extension.

image of El:

$$\begin{aligned}&\mathsf{data}\ \mathsf{in\text{-}U} : \mathsf{Set} \to \mathsf{Set}_1\\&\quad \mathsf{inBool} : \mathsf{in\text{-}U}\ \mathbb{2}\\&\quad \mathsf{inPi} : \{A : \mathsf{Set}\}\{B : A \to \mathsf{Set}\}\\&\qquad \to \mathsf{in\text{-}U}\ A \to ((a : A) \to \mathsf{in\text{-}U}\ (B\ a)) \to \mathsf{in\text{-}U}\ ((a : A) \to (B\ a))\end{aligned}$$

Using this U and El can be given as follows:

$$\begin{aligned}\mathsf{U} &= \Sigma(A : \mathsf{Set})\ (\mathsf{in\text{-}U}\ A)\\ \mathsf{El} &= \mathsf{proj}_1\end{aligned}$$

Note that this construction gives rise to a universe in Set_1, rather than Set, thus the metatheory must contain at least one universe. Our result is that a modified form of this translation also works for the more complex inductive-recursive type we need to model the universe of setoids. In particular, in addition to in-U for defining U as before, we also introduce a family in-U$\sim$ of binary relations between types in the universe, from which we then define $\_ \sim_{\mathcal{U}} \_$. However, we now need an inductive-inductive type in the target theory:

$$\begin{aligned}&\mathsf{data}\ \mathsf{in\text{-}U} : \mathsf{Set} \to \mathsf{Set}_1\\&\mathsf{data}\ \mathsf{in\text{-}U}\sim : \{A\ A' : \mathsf{Set}\} \to \mathsf{in\text{-}U}\ A \to \mathsf{in\text{-}U}\ A' \to (A \to A' \to \mathsf{Prop}) \to \mathsf{Set}_1\\&\mathcal{U} : \mathsf{Set}_1\\&\mathsf{El} : \mathcal{U} \to \mathsf{Set}\\&\_ \sim_{\mathcal{U}} \_ : \mathcal{U} \to \mathcal{U} \to \mathsf{Prop}_1\\&\_ \vdash \_ \sim_{\mathsf{El}} \_ : \{a\ a' : \mathsf{U}\} \to a\ \sim_{\mathcal{U}}\ a' \to \mathsf{El}\ a \to \mathsf{El}\ a' \to \mathsf{Prop}\\\\&\mathcal{U} = \Sigma\ (X : \mathsf{Set})\ (\mathsf{in\text{-}U}\ X)\\&\mathsf{El} = \mathsf{proj}_1\\&(X, p) \sim_{\mathcal{U}} (X', p') = \| \Sigma\ (R : X \to X' \to \mathsf{Prop})\ (\mathsf{in\text{-}U}\sim\ p\ p'\ R) \|\end{aligned}$$

where $\| A \| : \mathsf{Prop}$ denotes the propositional truncation of $A : \mathsf{Set}$. Intuition would suggest to define $\_ \vdash \_ \sim_{\mathsf{El}} \_$ by projecting the relation out of the proof of $a \sim_{\mathcal{U}} a'$, in much the same way as El is defined by projecting out of U. That is, $p \vdash x \sim_{\mathsf{El}} y = \mathsf{proj}_1\ p\ x\ y$. However, this doesn't work since the type of p is propositionally truncated, hence it cannot be used to construct a proof-relevant object. We can work around this by instead defining $\_ \vdash \_ \sim_{\mathsf{El}} \_$ by induction on the codes $a\ a' : \mathcal{U}$, explicitly for each type former, mutually with a proof that this definition is logically equivalent to what we would obtain if we could project out of the truncation.

We know that finitary inductive-inductive definitions can be translated into inductive families [AKKvR19, AKKvR18, KKA19] but it is not clear whether this construction extends to an infinitary type like the one above. This is subject of further work. If successful we would be able to give a translation of the setoid model with a universe into a very basic core type theory.

References

[ABKT19] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory—a syntactic translation. In Graham Hutton, editor, *Mathematics of Program Construction*, pages 155–196, Cham, 2019. Springer International Publishing.

[AKKvR18] Thorsten Altenkirch, Ambrus Kaposi, András Kovács, and Jakob von Raumer. Reducing inductive-inductive types to indexed inductive types. In José Espírito Santo and Luís Pinto, editors, *24th International Conference on Types for Proofs and Programs, TYPES 2018*. University of Minho, 2018.

[AKKvR19] Thorsten Altenkirch, Ambrus Kaposi, András Kovács, and Jakob von Raumer. Constructing inductive-inductive types via type erasure. In Marc Bezem, editor, *25th International Conference on Types for Proofs and Programs, TYPES 2019*. Centre for Advanced Study at the Norwegian Academy of Science and Letters, 2019.

[Alt99] Thorsten Altenkirch. Extensional equality in intensional type theory. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, page 412, USA, 1999. IEEE Computer Society.

[GCST19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, pages 1–28, January 2019.

[KKA19] Ambrus Kaposi, András Kovács, and Lafont Ambroise. For induction-induction, induction is enough. *Submitted to TYPES 2019 post-proceedings*, 2019.

The long exact sequence of homotopy n-groups

Ulrik Buchholtz[1] and Egbert Rijke[2]

[1] Technische Universität Darmstadt
buchholtz@mathematik.tu-darmstadt.de
[2] University of Ljubljana
egbert.rijke@fmf.uni-lj.si

An n-group G in homotopy type theory (Univalent Foundations Program, 2013) is defined to be (represented by) a pointed connected n-type BG (cf. Buchholtz, van Doorn, and Rijke, 2018). The *underlying type* G of the n-group BG is the loop space ΩBG, which possesses the structure of an n-group by the operations on paths. We note that the underlying type G of an n-group is $(n-1)$-truncated. The representing type BG of an n-group G is called the *classifying* type of the group.

From this point of view, a 1-group is a pointed connected 1-type. Loosely speaking, these are types that only have an interesting fundamental group, and no non-trivial higher homotopy groups.[1] The underlying type of a 1-group is therefore a set equipped with the usual structure of a group, so an ordinary group in the traditional sense of the word is a 1-group.

The principal class of examples of n-groups are the fundamental n-groups of pointed types. We define the fundamental n-group $\pi_1^{(n)}(X)$ of a pointed type X to be the n-truncation of the connected component of X at the base point.

Many n-groups G have further structure because they come with further deloopings of BG. The higher homotopy n-groups of a pointed type X are examples of such n-groups with additional symmetries. A k-symmetric n-group G is represented by a pointed $(k-1)$-connected $(n+k-1)$-type B^kG, of which the underlying type is the k-fold loop space $\Omega^k B^k G$. Note that a 1-symmetric n-group is just an n-group – group theory is about symmetries, after all. The notion of k-symmetric n-group stabilizes when $k \geq n+1$ (cf. Buchholtz, van Doorn, and Rijke, 2018, Sect. 6). For example, a 2-symmetric 1-group is an abelian group, and so is a k-symmetric 1-group for any $k \geq 2$.

The higher homotopy n-groups, $\pi_k^{(n)}(X)$, are now defined as the k-symmetric n-groups represented by

$$B^k\pi_k^{(n)}(X) :\equiv \|X\langle k-1\rangle\|_{n+k-1}.$$

In this definition of $\pi_k^{(n)}(X)$, the type $X\langle k-1\rangle$ is the $(k-1)$-connected cover of X, which is the fiber of the $(k-1)$-truncation of X:

$$X\langle k-1\rangle \longhookrightarrow X \longtwoheadrightarrow \|X\|_{k-1}.$$

Thus we see that $B^k\pi_k^{(n)}(X)$ fits in the fiber sequence

$$B^k\pi_k^{(n)}(X) \longhookrightarrow \|X\|_{n+k-1} \longtwoheadrightarrow \|X\|_{k-1},$$

and that the underlying type of $\pi_k^{(n)}(X)$ is equivalent to $\|\Omega^k X\|_{n-1}$.

[1] Of course, it is not quite as simple if there are noncontractible ∞-connected types around, as can happen if Whitehead's principle fails. Recall that homotopy type theory has models in $(\infty,1)$-toposes, and there are plenty such where Whitehead's principle fails.

The definition of the k'th homotopy n-group also makes sense when $k = 0$. In this case we just recover the $(n-1)$-truncation of X. The observation that $B^k\pi_k^{(n)}(X)$ is the fiber of $\|X\|_{n+k-1} \to \|X\|_{k-1}$ is a generalization of the well-known fiber sequence

$$K(\pi_k(X), k) \hookrightarrow \|X\|_k \twoheadrightarrow \|X\|_{k-1}$$

in which the fiber is the k'th Eilenberg-Mac Lane space of the k-th homotopy group of X (Licata and Finster, 2014).

The basic observation that we use to obtain the long exact sequence of homotopy n-groups is the following proposition, in which we establish that the n-truncation operation – although it is not left exact – preserves k-cartesian squares for any $k < n$. A square

$$\begin{array}{ccc} C & \longrightarrow & B \\ \downarrow & & \downarrow \\ A & \longrightarrow & X \end{array}$$

is called k-cartesian if the gap map $C \to A \times_X B$ is k-connected.

Proposition 1. *The n-truncation modality preserves k-cartesian squares for any $k < n$.*

In particular, any pullback square is $(n-1)$-cartesian, so the n-truncation of a pullback square is an $(n-1)$-cartesian square. We use this to obtain our main theorem

Theorem 2. *For any fiber sequence $F \hookrightarrow E \twoheadrightarrow B$ we obtain a long n-exact sequence*

$$\cdots \longrightarrow \pi_k^{(n)}(F) \longrightarrow \pi_k^{(n)}(E) \longrightarrow \pi_k^{(n)}(B) \longrightarrow \cdots \longrightarrow \pi_0^{(n)}(F) \longrightarrow \pi_0^{(n)}(E) \longrightarrow \pi_0^{(n)}(B)$$

of homotopy n-groups, where the morphisms are homomorphisms of k-symmetric n-groups whenever the codomain is a k-symmetric n-group.

1 Acknowledgements

The authors acknowledge the support of the Centre for Advanced Study (CAS) at the Norwegian Academy of Science and Letters in Oslo, Norway, which funded and hosted the research project Homotopy Type Theory and Univalent Foundations during the academic year 2018/19.

References

Buchholtz, Ulrik, Floris van Doorn, and Egbert Rijke (2018). "Higher Groups in Homotopy Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: ACM, pp. 205–214. DOI: 10.1145/3209108.3209150.

Licata, Daniel R. and Eric Finster (2014). "Eilenberg-MacLane spaces in homotopy type theory". In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, New York, Article No. 66, 10.

Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: http://homotopytypetheory.org/book/.

Categorical structures for type theory in univalent foundations, II

Benedikt Ahrens[1], Nikolai Kudasov[2], Peter LeFanu Lumsdaine[3], and Vladimir Voevodsky[4]

[1] University of Birmingham, UK
b.ahrens@cs.bham.ac.uk
[2] Innopolis University, Russia
n.kudasov@innopolis.ru
[3] Stockholm University, Sweden
p.l.lumsdaine@math.su.se
[4] Institute for Advanced Study, Princeton, NJ, USA
vladimir@ias.edu

1 Introduction

Various categorical structures have been introduced for studying type theories. In the present project, continuing [2], we compare several of these structures, working in univalent foundations.

Specifically, we compare categories with families, relative universes, and several variants of these, and investigate how they interact with univalence/saturation and the Rezk completion.

More generally, we explore the differences and novelties of studying algebraic structures in univalent foundations, compared to in classical foundations.

All our results have been formalised in Coq, over the UniMath library; specifically, in the tagged version `2020-AKLV-TYPES-abstract` of the `UniMath/TypeTheory` repository.

2 Comparing algebraic structures in the univalent setting

To meaningfully compare different kinds of structures in a classical setting, one must organise them into *categories*, and study *functors* between these categories. Equivalence of categories, for instance, gives a good notion of equivalence between two kinds of structures.

But the category structure is extra infrastructure that must be defined by hand. Off the shelf, the structures form just classes; and functions between these classes tell us little. Bijection between classes of structures, for instance, is not particularly meaningful or useful: it neither implies nor is implied by equivalence of the corresponding categories.

In the univalent setting, structures of some kind automatically form a *type*, which (thanks to its non-trivial equality types) carries much more information than the classical class of such structures. Typically, the type of widgets will correspond to the *groupoid core* of the category of widgets. (Precisely, this is univalence/saturation of the category of widgets.)

Equivalences of types of structures, or other properties of functions between these types, thus already give meaningful comparisons between the different kinds of structure.

In [2], we compared several different notions of structure at the level of types, giving functions between the types of such structures, and showing which of these functions are equivalences, embeddings, or surjections. In the present work, we raise these comparisons to the category level: we define (univalent) categories of these structures, and discuss how properties of functors between them correspond to the properties of the underlying functions.

3 Categorical structures for type theories

We mainly consider four types of structure: *categories with families*, *representable maps of presheaves*, *relative universes*, and *weak relative universes*.

A *category with families*, or *CwF* (Dybjer [4], as reformulated by Fiore [5] and Awodey [3]) consists of a category, whose objects are thought of as *contexts*, along with presheaves of *types* and *terms* connected by a map, and a *context extension* operation, characterised by a universal property. *Representable maps of presheaves* weaken this by just asserting existence of objects with the desired universal property, rather than an operation providing them.

Relative universes were introduced in [2]. They abstract away the role that presheaves play in the definition of CwF's: for a functor $J : \mathcal{C} \to \mathcal{D}$, a *$J$-relative universe* is a map in $\mathcal{D}$ together with an operation providing certain *J-pullbacks*. (For the Yoneda embedding $\mathsf{y}_{\mathcal{C}} : \mathcal{C} \to \mathsf{PreShv}(\mathcal{C})$, a $\mathsf{y}_{\mathcal{C}}$-relative universe is precisely a CwF structure on $\mathcal{C}$.) A *weak* J-relative universe is the same, but with just existence of suitable J-pullbacks, not a given operation.

4 Univalent categories and the Rezk-completion

A notable feature of category theory in univalent foundations (introduced by Ahrens, Kapulkin, and Shulman [1]) is that many categories of interest are *univalent* (also called *saturated*): equality of their objects corresponds precisely to isomorphism. Classically, this can only hold in degenerate cases; but in the univalent setting, it holds for most naturally constructed categories.

Working in univalent categories has various payoffs: since "isomorphism is equality", for instance, objects specified by universal properties become literally unique, and so existence conditions often imply (unique) existence of operations picking witnesses.

If a category $\mathcal{C}$ is not univalent, this can be rectified by the *Rezk-completion* construction, which performs a homotopy-quotient on the objects, replacing their original equality with the isomorphisms of $\mathcal{C}$, to give a new category $\mathsf{RC}(\mathcal{C})$, univalent and (weakly) equivalent to $\mathcal{C}$.

5 Summary

Our results are summarised by the following diagram of categories and functors:

$$\begin{array}{ccccccc}
\mathsf{SplTy}(\mathcal{C}) & \xleftarrow{\simeq} & \mathsf{Cwf}(\mathcal{C}) & \xleftarrow{\simeq} & \mathsf{RelU}(\mathsf{y}_{\mathcal{C}}) & \xrightarrow{\text{ff}} & \mathsf{RelU}(\mathsf{y}_{\mathsf{RC}(\mathcal{C})}) \xleftarrow{\simeq} \mathsf{Cwf}(\mathsf{RC}(\mathcal{C})) \\
 & & \downarrow{\scriptstyle \text{ff}} & & \downarrow{\scriptstyle \text{ff}} & & \uparrow{\scriptstyle \simeq} \qquad \uparrow{\scriptstyle \simeq} \\
 & & \mathsf{Rep}(\mathcal{C}) & \xleftarrow{\simeq} & \mathsf{RelWkU}(\mathsf{y}_{\mathcal{C}}) & \xleftarrow{\simeq} & \mathsf{RelWkU}(\mathsf{y}_{\mathsf{RC}(\mathcal{C})}) \xleftarrow{\simeq} \mathsf{Rep}(\mathsf{RC}(\mathcal{C}))
\end{array}$$

Here, $\mathsf{SplTy}(\mathcal{C})$ is the category of split type-category structures on a base category $\mathcal{C}$; $\mathsf{Cwf}(\mathcal{C})$ the category of CwF structures on $\mathcal{C}$; $\mathsf{RelU}(F)$ (resp. $\mathsf{RelWkU}(J)$) the category of (weak) J-relative universes, for a functor J; and $\mathsf{Rep}(\mathcal{C})$ the category of representable maps of presheaves on $\mathcal{C}$.

Acknowledgments

The work reported here was planned and begun in 2017 by Ahrens, Lumsdaine, and Voevodsky, as a sequel to [2]. Sadly, Voevodsky died unexpectedly in August 2017; the project was resumed in 2019 in collaboration with Kudasov. The remaining authors are grateful to Daniel R. Grayson, Vladimir's academic executor, for his advice and support in preparing this work.

References

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

[2] Benedikt Ahrens, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. Categorical structures for type theory in univalent foundations. *Logical Methods in Computer Science*, 14(3), 2018.

[3] Steve Awodey. Natural models of homotopy type theory. *Mathematical Structures in Computer Science*, pages 1–46, 2016.

[4] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995.

[5] Marcelo Fiore. Discrete generalised polynomial functors, 2012. Slides from talk given at ICALP 2012, http://www.cl.cam.ac.uk/~mpf23/talks/ICALP2012.pdf.

Natural Numbers in Homotopy Type Theory

Marco Benini and Roberta Bonacina

Dipartimento di Scienza e Alta Tecnologia,
Università degli Studi dell'Insubria, Italy
marco.benini@uninsubria.it r.bonacina1@uninsubria.it

1 Introduction

The type $\mathbb{N}$ of natural numbers is an integral part of intensional Martin-Löf's type theory [2, 3]. Quoting Section 1.9 of [5],

> The rules we have introduced so far do not allow us to construct any infinite types. The simplest infinite type we can think of (and one which is of course also extremely useful) is the type $\mathbb{N} : \mathcal{U}_i$ of natural numbers.

This assertion is certainly true in the book, as it is introducing type theory from scratch, and historically well founded, since the problem of having an infinite object comes from set theory, and it has been solved defining natural numbers (in fact, the ω ordinal) as a primitive entity. However, it can and should be challenged in homotopy type theory where spaces, paths, and homotopies are the fundamental notions. This is exactly the purpose of this contribution: to show that, from a purely foundational point of view, the type $\mathbb{N}$ can be derived from the 1-sphere, which is a more natural primitive type within the homotopy framework. In [4] the same problem has been addressed and solved by a convoluted approach. In the present contribution, a conceptually simpler solution is provided.

It is worth remarking that this is not the only way to encode natural numbers from another canonical type; indeed it can be done with W-types, as illustrated in Section 5.5 of [5].

2 Integers and the 1-sphere

The higher inductive type $\mathbb{S}^1$ (Section 6.1 in [5]) is generated by a point $\mathsf{base} : \mathbb{S}^1$ and a path $\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$. The induction principle derives from $P : \mathbb{S}^1 \to \mathcal{U}$, $b : P(\mathsf{base})$, and $\ell : b =^P_{\mathsf{loop}} b$ that there is $f : \Pi_{x:\mathbb{S}^1} P(x)$ such that $f(\mathsf{base}) \equiv b$ and $f(\mathsf{loop}) \equiv \ell$. In Section 8.1 of [5], and originally in [1], it has been shown that $\Omega(\mathbb{S}^1, \mathsf{refl}_{\mathsf{base}})$, the *loop space* of the 1-sphere is equal to the type of integer numbers. We make the bold move to *define* $\mathbb{Z} :\equiv \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$, that is, $\mathbb{Z}$ is identified with the loop space $\Omega(\mathbb{S}^1)$.

Calling $0 :\equiv \mathsf{refl}_{\mathsf{base}}$, $1 :\equiv \mathsf{loop}$, $-1 :\equiv \mathsf{loop}^{-1}$, $+$ path concatenation, the result shows that each path in $\mathbb{Z}$ is equal to one of the *canonical* forms $\mathsf{refl}_{\mathsf{base}}$, loop^n, and loop^{-n}, the finite (n times) concatenation of loop and loop^{-1} with itself. Then, there exists $\mathsf{eval} : \mathbb{Z} \to \mathbb{Z}$ such that $\mathsf{eval}(x) = x$ and $\mathsf{eval}(x)$ is a canonical form: in fact, eval is the quasi-inverse of the identity in the equivalence $\Omega(\mathbb{S}^1) \simeq \Omega(\mathbb{S}^1)$. The existence of eval is a sort of local normalisation theorem, which shows that each path in $\mathbb{S}^1$ can be reduced to a canonical one. Then, after [1], it is easy to derive the usual introduction, elimination, and computation rules for the integer numbers from the rules of $\mathbb{S}^1$. However, notice how the computation rules use the propositional equality instead of the judgemental one, as they are derived from a higher inductive type: in fact, $\mathbb{Z}$ *is* a higher inductive type in this setting.

Moreover, it suffices to define the usual functions (addition, multiplication, etc.) on the canonical loops to extend them to all terms of type $\mathbb{Z}$. Their definition is the usual one as the recursion principle of $\mathbb{Z}$ holds.

3 Naturals from integers

Then, a neat way to characterise naturals from integers is to define the absolute value function: the obvious map abs defined by $0 \mapsto 0$, $1 \mapsto 1$, $-1 \mapsto 1$ does not extend to $+$ for a generic term $t : \mathbb{Z}$, because $1 + -2 \mapsto 3$ by a direct application of the map, while $1 + -2 \mapsto 1$ noticing that $1 + -2 =_{\mathbb{S}^1} -1$. However, abs can be easily defined on canonical loops: it maps $0 \mapsto 0$, $\mathsf{loop}^n \mapsto \mathsf{loop}^n$, and $\mathsf{loop}^{-n} \mapsto \mathsf{loop}^n$. Extending abs to $\mathbb{Z}$ yields $|\_| : \mathbb{Z} \to \mathbb{Z}$, the sought absolute value function: the extended function is, in fact, just $\mathsf{abs} \circ \mathsf{eval}$.

Then, natural numbers can be defined as the subtype of $\mathbb{Z}$ inhabited by the fix points of $|\_|$, that is $\mathbb{N} :\equiv \Sigma_{x:\mathbb{Z}}\, x =_{\mathbb{Z}} |x|$. Since $0 : \mathbb{N}$, and if $n : \mathbb{N}$ then $\mathsf{succ}(n) : \mathbb{N}$, the $\mathbb{N}$-introduction rules are justified. The induction principle on naturals is an immediate instance of the $\mathbb{Z}$-elimination rule, noticing how the pred premise becomes trivial on canonical loops (there are no loops of the form loop^{-n} in $\mathbb{N}$). Finally, the computation rules for $\mathbb{N}$ are immediately inherited from those of $\mathbb{Z}$, but they are expressed using propositional equality. In other words, $\mathbb{N}$ as defined here, is a higher inductive type.

Clearly, addition and multiplication are inherited from $\mathbb{Z}$ because of the properties of the absolute vale, thus the usual axioms of Peano arithmetic can be derived.

The interest of this construction is that from a purely foundational point of view, it shows that there is no need to define $\mathbb{N}$ as a standalone type since it can be derived from $\mathbb{S}^1$ (but we have to give up judgemental equality in favour of propositional equality). In the perspective of the homotopy interpretation, we see how integers and naturals are derived entities from a natural homotopy space, the 1-sphere. Finally, the existence of eval strictly relates this contribution with the interpretation of homotopy type theory as an abstract programming language, as it allows to map arithmetical expressions to their value.

References

[1] D.R. Licata and M. Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 223–232, June 2013.

[2] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975.

[3] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.* Bibliopolis, Napoli, 1984.

[4] Robert Rose. Constructing $\mathbb{N}$ by $\mathbb{S}^1$ induction. Talk at Homotopy Type Theory, 2019.

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

On Symmetries of Spheres in HoTT-UF

Pierre Cagne* Nicolai Kraus† Marc Bezem*

The goal of this talk is to give insights in the symmetries of the n-sphere in synthetic homotopy theory, i.e. the type $S^n = S^n$.

We work in intuitionistic Martin-Löf's type theory with Σ-, Π- and Id-types and with a cumulative hierarchy of universes, simply written $\mathbb{U}$, for which Voevodsky's univalence axiom hold. A good reference for our setting is the HoTT Book [Uni13], to which we will refer frequently. We specifically use the following higher inductive types (HITs): the propositional truncation $\|A\|$ of an arbitrary type A ([Uni13, Ch. 3.7]); the set truncation $\|A\|_0$ of an arbitrary type A ([Uni13, Ch. 6.9]); the circle S^1 ([Uni13, Ch. 6.4]); the suspension ΣA of an arbitrary type A ([Uni13, Ch. 6.5]).

We give some more details of the circle and the suspension, as they are crucial for our presentation. The circle is a higher inductive type with one point constructor and one path constructor:

$$\begin{aligned} \bullet &: S^1, \quad \text{the base point} \\ \circlearrowleft &: \bullet =_{S^1} \bullet, \quad \text{the loop} \end{aligned}$$

The circle comes with an elimination rule such that functions $S^1 \to A$ correspond to pairs of an element $a : A$ and a path $\ell : a = a$ for any type A. The first part of the talk will be dedicated to showing that

$$(S^1 = S^1) \simeq (S^1 + S^1). \tag{1}$$

For $n \geq 2$, the n-sphere S^n is inductively defined as the suspension $\Sigma(S^{n-1})$. The suspension ΣA of a type A is a higher inductive type with two point constructors and path constructors indexed by A:

$$\begin{aligned} N &: \Sigma A, \quad \text{the 'North pole'} \\ S &: \Sigma A, \quad \text{the 'South pole'} \\ \mathit{merid} &: A \to (N =_{\Sigma A} S), \quad \text{the 'meridians'} \end{aligned}$$

The elimination principle for ΣA gives again a correspondence between the type $\Sigma A \to B$ and the type of triplets consisting of $b_N : B$, $b_S : B$ and $m : A \to b_N = b_S$.

One cannot expect (1) to generalise to higher dimensions. However, (1) implies that $S^1 = S^1$ consists of two equivalent connected components. Modulo univalence, one component contains id_{S^1} and the other $-\mathrm{id}_{S^1}$. The latter is the function $S^1 \to S^1$ corresponding to the pair $\bullet : S^1$ together with the path $\circlearrowleft^{-1} : \bullet = \bullet$. This weaker statement does in fact generalise to higher

*Universitetet i Bergen
†University of Birmingham

spheres in the homotopy theory of topological spaces. In the talk, we will elaborate a proof in HoTT-UF for the case $n = 2$, along the following lines.

The first step is to define $-\mathrm{id}_{\Sigma A}$ for any type A as the function corresponding to the triplet $S : \Sigma A$, $N : \Sigma A$ and $merid(\_)^{-1} : (A \to S = N)$. In other words, $-\mathrm{id}_{\Sigma A}$ flips the poles and reverses each meridian. Notice that $-\mathrm{id}_{\Sigma A}$ is an equivalence, as it is its own pseudo-inverse. The function

$$\mathrm{flip}_A :\equiv \_ \circ -\mathrm{id}_{\Sigma A} : (\Sigma A \to \Sigma A) \to (\Sigma A \to \Sigma A)$$

is then an equivalence, hence establishing an equivalence from the connected component at $\mathrm{id}_{\Sigma A}$ to the connected component at $-\mathrm{id}_{\Sigma A}$. Notice that $-\mathrm{id}_{\Sigma A}$ is not necessarily distinct from $\mathrm{id}_{\Sigma A}$ (take for example $A :\equiv 1$ for which ΣA is contractible). In particular, it is non-trivial to prove that $-\mathrm{id}_{S^2} \neq \mathrm{id}_{S^2}$.

As the sphere S^2 is connected, in proving the proposition $\|\varphi = \mathrm{id}_{S^2}\| + \|\varphi = -\mathrm{id}_{S^2}\|$ for an equivalence $\varphi : S^2 \simeq S^2$, one can as well suppose that φ is a pointed map by a path $\varphi_0 : N = \varphi(N)$. It is worth computing the degree of such a φ. Recall that the degree $d(f, f_0)$ of a pointed function $(f, f_0) : S^2 \to_* S^2$ is defined as the integer $\bar{\pi}_2(f, f_0)(1)$ where $\bar{\pi}_2(f, f_0)$ is the group morphism $\pi_2(f, f_0) : \pi_2(S^2) \to \pi_2(S^2)$ viewed through the equivalence $\pi_2(S^2) \simeq \mathbb{Z}$.

For example, for each $k : \mathbb{Z}$, the following map $\delta_k : S^2 \to_* S^2$ has degree k: first define $c_k : S^1 \to S^1$ as the map that corresponds to the pair $\bullet : S^1$ together with the path $\circlearrowleft^k : \bullet = \bullet$; then define δ_k as the map corresponding to the triplet $N : S^2$, $S : S^2$ and $merid \circ c_k : S^1 \to N = S$, obviously pointed by the path $refl_N : N = \delta_k(N)$. It is easy to see that $\delta_1 = \mathrm{id}_{S^2}$ and one can prove that $\delta_{-1} = -\mathrm{id}_{S^2}$ also. Proving that δ_k has indeed degree k is non-trivial, and we shall exhibit a proof using the Hopf fibration.

Using the functoriality of π_2, one gets

$$d((g, g_0) \circ (f, f_0)) = \bar{\pi}_2(g, g_0)\,(\bar{\pi}_2(f, f_0)(1)) = \bar{\pi}_2(g, g_0)(1) \times \bar{\pi}_2(f, f_0)(1).$$

The last identity comes from the fact that $\bar{\pi}_2(g, g_0)$ is a group morphism $\mathbb{Z} \to \mathbb{Z}$. In other words, d is a morphism of monoids, and as such, it maps equivalences to invertible elements of $\mathbb{Z}$. Hence, $d(\varphi, \varphi_0) = \pm 1$. The last step is to prove that having the same degree means precisely being in the same connected component of $S^2 \to_* S^2$. In order to do so, we shall give an alternate description of the degree, based on the Hopf fibration, and on the proof that $\pi_2(S^2) \simeq \mathbb{Z}$ (cf. [Uni13, Ch. 8.6]).

This result generalizes to the case $n > 2$ with the help of the Freudenthal suspension theorem ([Uni13, Ch. 8.6]). If time permits, we will sketch our path to a full proof of the fact that $S^n = S^n$ has exactly two connected components.

Future works include formalizing this proof in cubical type theory (CTT) and experimenting with actual computation of the degree of selected symmetries. This is one of the motivations for this work. Indeed, the univalence axiom is necessary for the definition of the Hopf fibration and for the Freudenthal suspension theorem (and hence for the definition of the degree), and an implementation in CTT will display the computational content of univalence at work. Hopefully, this would be a feasible computational challenge, simpler than the computation of Brunerie's number (cf. [Bru16, Corollary 3.4.5]), which is still out of reach of CTT and other systems.

[Bru16] Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory.* PhD thesis, Université de Nice Sophia Antipolis, June 2016.

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

2 Induction and identity in higher types (WG1)

A model of type theory with quotient inductive-inductive types *

Ambrus Kaposi and Zongpu Xie

Eötvös Loránd University, Budapest, Hungary
akaposi@inf.elte.hu and szumixie@gmail.com

QIITs. Quotient inductive-inductive types (QIITs) are a general class of inductive types that allow multiple sorts indexed over each other (inductive-inductive [10]) and support equality constructors (quotient). An example is given in the left column of Figure 1. This QIIT has two sorts, Con and Ty, five point constructors $\bullet$, ..., Σ, and an equality constructor eq. It comes with two elimination principles (one for each sort, we don't list them here) which enforce that every function from Con preserves the equality eq. Con–Ty can be extended to the full syntax of type theory [4]. Other examples of QIITs include the real numbers [11] and the partiality monad [3]. Kaposi et al. [8] gave a general definition of QIITs and showed that all finitary QIITs can be constructed from a single QIIT called the universal QIIT. However they did not show that this universal QIIT exists.

In this talk we show that there is a model of type theory which supports the universal QIIT, namely the setoid model [1]. The setting of [8] is extensional type theory, hence by the conservativity result of Hofmann [7] the construction can be transferred to a model of type theory with function extensionality and uniqueness of identity proofs (UIP). As these hold in the setoid model, we conclude that all finitary QIITs can be defined in the setoid model.

The contents of this talk were formalised in Agda, we provide links to specific parts below.

Con	$:\mathsf{Set}$	Con^s	$:\mathsf{Ty}\ \bullet$
Ty	$:\mathsf{Con}\to\mathsf{Set}$	Ty^s	$:\mathsf{Ty}\ (\bullet\triangleright\mathsf{Con}^\mathsf{s})$
$\bullet$	$:\mathsf{Con}$	$\bullet^\mathsf{s}$	$:\mathsf{Tm}\ \bullet\ \mathsf{Con}^\mathsf{s}$
$-\triangleright-$	$:(\Gamma:\mathsf{Con})\to\mathsf{Ty}\ \Gamma\to\mathsf{Con}$	$\triangleright^\mathsf{s}$	$:\mathsf{Tm}\ (\bullet\triangleright\mathsf{Con}^\mathsf{s}\triangleright\mathsf{Ty}^\mathsf{s})\ (\mathsf{Con}^\mathsf{s}\,[\epsilon])$
U	$:\mathsf{Ty}\ \Gamma$	U^s	$:\mathsf{Tm}\ (\bullet\triangleright\mathsf{Con}^\mathsf{s})\ \mathsf{Ty}^\mathsf{s}$
El	$:\mathsf{Ty}\ (\Gamma\triangleright\mathsf{U})$	El^s	$:\mathsf{Tm}\ (\bullet\triangleright\mathsf{Con}^\mathsf{s})\ (\mathsf{Ty}^\mathsf{s}\,[\epsilon,\triangleright^\mathsf{s}\,[\mathsf{id},\mathsf{U}^\mathsf{s}]])$
Σ	$:(A:\mathsf{Ty}\ \Gamma)\to\mathsf{Ty}\ (\Gamma\triangleright A)\to\mathsf{Ty}\ \Gamma$	Σ^s	$:\mathsf{Tm}\ (\bullet\triangleright\mathsf{Con}^\mathsf{s}\triangleright\mathsf{Ty}^\mathsf{s}\triangleright\mathsf{Ty}^\mathsf{s}\,[\epsilon,\triangleright^\mathsf{s}])\ (\mathsf{Ty}^\mathsf{s}\,[\mathsf{wk}^2])$
eq	$:\Gamma\triangleright\Sigma\ A\ B=\Gamma\triangleright A\triangleright B$	eq^s	$:\mathsf{Tm}\ (\bullet\triangleright\mathsf{Con}^\mathsf{s}\triangleright\mathsf{Ty}^\mathsf{s}\triangleright\mathsf{Ty}^\mathsf{s}\,[\epsilon,\triangleright^\mathsf{s}])$
			$(\mathsf{El}\ (\mathsf{Id}\ (\mathsf{Con}^\mathsf{s}\,[\epsilon])\ (\triangleright^\mathsf{s}\,[\mathsf{wk}^2,\Sigma^\mathsf{s}])\ (\triangleright^\mathsf{s}\,[(\epsilon,\triangleright^\mathsf{s})^\uparrow])))$

Figure 1: Constructors of the QIIT Con–Ty, a fragment of the well-typed syntax of type theory. Note that Con, Ty on the left become Con^s, Ty^s on the right and Ty, Tm on the right are those of a model of type theory.

Specification of QIITs in a model. We use categories with families (CwFs [5]) as the notion of model of type theory. That is, a model is a category given by objects Con, morphisms Sub, families Ty, Tm, substitution is written $-[-]$, empty context $\bullet$, context extension $\triangleright$.

*The first author was supported by the ÚNKP-19-4 New National Excellence Program of the Ministry for Innovation and Technology and by the Bolyai Fellowship of the Hungarian Academy of Sciences. The second author was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Specifying what it means for a model to support a QIIT is straightforward but tedious. For the constructors of Con–Ty this is done on the right hand side of Figure 1. Here Ty and Tm refer to types and terms of the model, not sorts of the QIIT. The elimination principle can be specified in a similar way. The difficulty of working inside a model is that we have to write out all arguments explicitly (e.g. Γ for U), we need weakenings (e.g. in the type of Σ^{s}) and CwF-combinators instead of variables names. For modularity and efficiency reasons we don't use function space of the model to list parameters of constructors, instead we add them to the context. E.g. $\triangleright^{\mathsf{s}}$ is not a function with two arguments, but a term in a context of length two.

The universal QIIT is a syntax for a small type theory describing signatures of QIITs. It is (roughly) an extension of our Con–Ty example. We specified the constructors and elimination principles of the universal QIIT. These specifications import the setoid model, but they also work with the standard (set) model or any other strict model. Importing the standard model and normalising the types helps to make sure that the internal specification (right hand side of Figure 1) corresponds to the external one (left hand side of the figure).

The setoid model. In the setoid model [1], an element of Con is a setoid, that is, a set with a (Prop-valued) equivalence relation. The idea is that each type comes with its own identity type encoded in this relation. For example, the relation for function space says that the two functions are extensionally equal. This way the setoid model supports function extensionality, propositional extensionality and quotients. We formalised the setoid model in Agda without using any axioms or postulates, we also did not rely on UIP in Agda. The only special feature that we used was the definitionally proof irrelevant universe of propositions Prop [6]. The setoid model is strict [2], that is, all the equalities are definitional in Agda.

Implementation of QIITs in the setoid model. We defined an IIT with four sorts which we call the *implementation IIT* for Con–Ty. There are the two Set-sorts for Con and Ty and two Prop-sorts for their equality relations. The constructor eq is a constructor of the equality relation for Con. The Set-sort for Ty includes an additional coercion constructor, the Prop-sorts include constructors expressing that they are equivalence relations, congruence rules and that coercion respects the relation. With the help of the implementation IIT, we defined the constructors for Con–Ty. The elimination principle is defined simply by pattern matching (eliminating from the implementation IIT). We defined both the recursion principle with uniqueness and the dependent elimination principle for Con–Ty. All the computation rules hold definitionally.

The above method also works for the universal QIIT, we defined the recursor and formalised uniqueness. We haven't managed to typecheck uniqueness yet due to performance problems.

Further work. We would like to extend the universal QIIT with more type formers to allow non-closed QIITs (metatheoretic Π), infinitary constructors, equalities as inputs of constructors, sort equalities. We formalised that the setoid model supports arbitrary branching trees where the order of subtrees do not matter. This is an infinitary QIT which seems not to be definable from quotients without using the axiom of choice [4]. In fact, some QITs are known not to be constructible without the axiom of choice [9, Section 9].

We showed that finitary QIITs exist in the setoid model, but can they be defined in the set model (using only quotients)? There is a weak morphism of models from the setoid model to the set model, we plan to investigate what this morphism maps the universal QIIT to.

As the setoid model is given in an intensional metatheory, it provides a computational interpretation of the QIITs we defined. It remains to be checked what happens if we replay the construction of other QIITs from the universal QIIT [8]. Would we still get definitional computation rules?

References

[1] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[2] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory—a syntactic translation. In Graham Hutton, editor, *Mathematics of Program Construction*, pages 155–196, Cham, 2019. Springer International Publishing.

[3] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures - Volume 10203*, page 534–549, Berlin, Heidelberg, 2017. Springer-Verlag.

[4] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodik and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016.

[5] Peter Dybjer. Internal type theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.

[6] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, pages 1–28, January 2019.

[7] Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *TYPES 95*, pages 153–164, 1995.

[8] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL):2:1–2:24, January 2019.

[9] Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types, 2012. Note.

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

[11] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

A Cubical Approach to the Structure Identity Principle

Anders Mörtberg and Max Zeuner

Stockholm University, Stockholm, Sweden
{anders.mortberg,zeuner}@math.su.se

Homotopy Type Theory/Univalent Foundations (HoTT/UF) is a new approach to the foundations of mathematics that combines intensional Martin-Löf type theory (MLTT) with homotopy theory. One of the most remarkable new features of this theory is the univalence axiom, which was first introduced by Voevodsky in [8], that makes reasoning in HoTT/UF invariant under equivalence: given two types X and Y such that $X \simeq Y$ (i.e. X and Y are *equivalent* in some appropriate sense) and some type-theoretic statement $\Phi(X)$, then if $\Phi(X)$ is provable in HoTT/UF, so is $\Phi(Y)$.

Informally this desirable property is more generally expressed by the so-called *structure identity principle* (SIP), which asserts that reasoning about mathematical structures is invariant under isomorphisms of such structures. This can be made precise in HoTT/UF and formalized versions of the SIP have been proved for large classes of mathematical structures. The first type-theoretic formalization and proof of the SIP for algebraic structures was given by Coquand and Danielsson in [3]. A more general version, relying on the notion of univalent categories, can be found in section 9.8 of the HoTT-book [6]. For recent work on a higher SIP, generalizing the categorical SIP from the HoTT-book, see Ahrens et. al. [1]. Yet another variation, that more closely resemble the Coquand-Danielsson SIP than the one in the HoTT-book in the sense that it is more type-theoretic than categorical, can be found in the lecture notes of Escardó [5]. The cubical SIP that we will discuss in this abstract is mainly inspired by Escardó's variation and our work can be seen as a reformulation of it in cubical type theory.

In all of the aforementioned versions of the SIP the univalence axiom plays a crucial role. However, one caveat of the univalence axiom is that it breaks the computational character of type theory so that the SIP in HoTT/UF lacks computational content. To remedy this, cubical type theory was developed by Coquand et. al. [2]. Cubical type theory adds new primitives to MLTT, in particular an interval which makes it possible to replace the identity type by path types represented as functions out of the interval. The univalence axiom is then provable in cubical type theory using so-called Glue-types, giving it computational content. Recently, `Agda` was extended with a *cubical mode* [4] making it possible to work in cubical type theory using a fully fledged proof assistant [7]. We write `ua` for the function underlying the univalence axiom in `Cubical Agda`:

$$\mathsf{ua} \;:\; \{A\ B \;:\; \mathcal{U}\} \to A \simeq B \to \mathsf{Path}_{\mathcal{U}}(A, B)$$

By developing the SIP in `Cubical Agda` we obtain a SIP with computational content that allows us to transport programs and proofs not just between equivalent types but also between equivalent *structures* on these types. Furthermore, the transported programs and proofs will all have computational content which simplifies both programming and proving. Our formalization of the cubical SIP can be found at:

https://github.com/agda/cubical/blob/master/Cubical/Foundations/SIP.agda

Following Escardó, we formalize structures as functions on a universe $S : \mathcal{U} \to \mathcal{U}$. Such a structure S comes equipped with a criterion telling us which equivalences of the underlying types of two S-structures A and B are "structure-preserving" isomorphisms

$$\iota : \left\{A\ B \;:\; \sum_{X\,:\,\mathcal{U}} S(X)\right\} \to \left(\mathtt{fst}\ A \;\simeq\; \mathtt{fst}\ B\right) \to \mathcal{U}$$

The type of isomorphisms between A and B is thus given by:

$$A \cong B \;:\equiv \sum_{e\colon \mathtt{fst}\ A \simeq \mathtt{fst}\ B} \iota(e)$$

Using path types, the cubical SIP that we aim to prove can then be formulated as:

$$(A \cong B) \;\simeq\; \mathsf{Path}_{\sum_{X\colon \mathcal{U}} S(X)}\ (A, B)$$

In order to be able to prove the SIP we need to formulate a criterion when such a pair $(S,\ \iota)$ of structures and isomorphisms defines a *standard notion of structure*, i.e. when such a pair is well behaved enough for the SIP to hold. We say that $(S,\ \iota)$ defines a *standard notion of structure*, if for all $X\ Y : \mathcal{U}$, $s : S(X)$, $t : S(Y)$ and $e : X \simeq Y$ we have an equivalence

$$\mathsf{Path}_{S(Y)}\ \big(\mathsf{transport}\ \big(\lambda i.\ S\ (\mathsf{ua}\ e\ i)\big)\ s\ ,\ t\big) \;\simeq\; \iota(e)$$

This definition of a standard notion of structure is equivalent to Escardó's original definition, but it interacts better with the cubical machinery of Glue-types. Because of this we can use Glue-types to directly define, for a standard notion of structure $(S,\ \iota)$, an equivalence

$$\mathsf{sip} :\ (A \cong B) \simeq \mathsf{Path}_{\sum_{X:\mathcal{U}} S(X)}\ (A, B)$$

The function underlying this equivalence gives direct computational meaning to the SIP in terms of the primitives of cubical type theory.

Applications of the cubical structure identity principle

Following Escardó, we show that adding proposition-valued axioms to a structure S as well as combining two structures $(S_1,\ \iota_1)$ and $(S_2,\ \iota_2)$ both preserve being a standard notion of structure. Interestingly, the cubical proofs of these facts are more direct than the corresponding ones in HoTT/UF.

Finally, we apply the cubical SIP to the concrete example of monoids. Given a type $X : \mathcal{U}$ the type of monoid structures is given by

$$\mathsf{monoid\text{-}structure}(X) \;:\equiv \sum_{e\ :\ X}\ \sum_{\_\cdot\_\ :\ X\to X\to X} \mathsf{isSet}(X)\ \times\ \mathsf{monoid\text{-}axioms}(X,\ e,\ \_\cdot\_)$$

where $\mathsf{monoid\text{-}axioms}(X,\ e,\ \_\cdot\_)$ is the product of the type theoretic formalization of the monoid axioms. The type of monoids (in $\mathcal{U}$) is then given by

$$\mathsf{Monoids} \;:\equiv \sum_{X\colon \mathcal{U}} \mathsf{monoid\text{-}structure}\ (X)$$

For two monoids $M\ N :\ \mathsf{Monoids}$ the type of monoid-isomophisms is given by

$$M \cong N \;:\equiv \sum_{\varphi\colon \mathtt{fst}\ M \simeq \mathtt{fst}\ N} \mathsf{Path}_{\mathtt{fst}\ N}\big(\varphi(e_M),\ e_N\big) \times \Big(\prod_{x\ y\colon \mathtt{fst}\ M} \mathsf{Path}_{\mathtt{fst}\ N}\big(\varphi(x\cdot y),\ \varphi(x)\cdot\varphi(y)\big)\Big)$$

Following Escardó, we show that monoid structures are a standard notion of structure by observing that they are obtained by combining simpler structures (i.e. pointed types and ∞-magmas) and adding proposition-valued axioms. The cubical SIP then allows us to prove

$$\big(M\ N :\ \mathsf{Monoids}\big) \;\to\; \big(M \cong N\big) \;\simeq\; \mathsf{Path}_{\mathsf{Monoids}}(M\ ,\ N)$$

which in turn lets us transport programs and proofs between isomorphic monoids without sacrificing the computational content of the transported programs and proofs.

References

[1] Benedikt Ahrens, Paige R. North, Michael Shulman, and Dimitris Tsementzis. A higher structure identity principle. https://benediktahrens.net/talks/BA-HSIP-FAUM-2019.pdf, dec 2019. Slides from FAUM 2019, https://cj-xu.github.io/faum/.

[2] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. *IfCoLog Journal of Logics and their Applications*, 4(10):3127–3169, November 2017.

[3] Thierry Coquand and Nils Anders Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105–1120, 2013.

[4] Agda developers. Agda 2.6.0.1 documentation. https://agda.readthedocs.io/en/v2.6.0.1/, 2019.

[5] Martín Hötzel Escardó. Introduction to univalent foundations of mathematics with agda. https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/index.html, 2019.

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

[7] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3:1–29, jul 2019.

[8] Vladimir Voevodsky. The equivalence axiom and univalent models of type theory. (Talk at CMU on February 4, 2010). Preprint arXiv:1402.5556, 2014.

Higher categories of algebras for higher inductive definitions

Paolo Capriotti[1] and Christian Sattler[2] *

[1] Darmstadt Technical University, Darmstadt, Germany
paolo@capriotti.io
[2] University of Nottingham, Nottingham, United Kingdom
sattler.christian@gmail.com

Abstract

We present an interpretation of the theory of signatures introduced in [6] based on Segal types in homotopy type theory. As a consequence, we obtain a construction of an internal higher category of algebras for an arbitrary definition of a higher inductive(-inductive) type. We show its finite completeness and use it to prove the equivalence between the induction principle and the initiality property.

1 Modelling signatures via Segal types

Based on the corresponding notion in [6, 7], let us define a *signature model* as a CwF equipped with a universe $\mathcal{U}$ closed under equality types, and Π-types with domain in $\mathcal{U}$. The theory of signatures is the initial signature model.[1] A *signature* for a higher inductive-inductive type is then simply a context in the theory of signatures.

We construct a signature model in (marked) semisimplicial types on an arbitrary model $\mathcal{C}$ of homotopy type theory. Since it is not known how to work with semisimplicial types internally, we use two-level type theory ([2, 3]) as an internal language of presheaves over $\mathcal{C}$. Following the main ideas of [5], we augment semisimplicial types with a type of "markings" on the edges. This makes it possible to express both Segal and completeness conditions in terms of lifting properties.

We distinguish several classes of fibrations, corresponding to different choices of lifting properties. The most general are Reedy fibrations, as in [8, 3]. To obtain (internal) higher categorical semantics, we also consider *inner fibrations*, i.e. maps orthogonal to inner horn inclusions, giving a very general notion of "dependent higher category". To capture the correct notion of algebra morphism (rather than algebra relation), we must also consider *left fibrations*, corresponding to covariant families of types.

Our model is as follows: contexts are marked semisimplicial types, types are relative complete Segal types, i.e. Reedy fibrant presheaves over the category of elements of the base context such that the corresponding display map is an inner isofibration satisfying a completeness condition. The universe is set to be a classifying presheaf for left fibrations, defined as a subpresheaf of the standard universe in the Reedy model [8, 3] consisting of those simplices corresponding to left fibrations. Finally, Π-types and equality types are interpreted as in the Reedy model.

Since left fibrations are in particular exponentiable [4], it follows that the Π-type of a left fibration and an inner fibration is again an inner fibration. Closure of left fibrations under equality types follows by a cancellation property of left fibrations. Finally, the classifier for left

*Supported by USAF grant FA9550-16-1-0029.

[1] The theory of signatures in [6] contains more type formers: it allows parameters from an ambient model of homotopy type theory for both type signatures and constructors in signatures for higher inductive types. In this abstract, we use a reduced version for the sake of simplifying the presentation. Our results apply to the original notion of [6], optionally strengthened with judgmental β-laws for the identity type in $\mathcal{U}$.

fibrations models the higher category of small types (with respect to the chosen universe), in particular is a complete Segal type.

2 Equivalence of initiality and induction

For any signature Γ, we want to think of an initial object of the corresponding higher category of algebras $\mathcal{A}$ as the higher inductive-inductive type defined by Γ. In other words, we want to relate the initiality property in $\mathcal{A}$ with the induction principle associated to Γ.

This can be done in two steps. First, similarly to [1], we introduce the notion of *section initiality*: an object H in $\mathcal{A}$ is section-initial if for all objects X of $\mathcal{A}$, every morphism $H \to X$ has a section. Then a well-known categorical argument implies that if $\mathcal{A}$ has finite limits, section-initiality and initiality coincide. After that, we relate section-initiality of H to the induction principle for H. The latter is obtained by interpreting the theory of signatures into the model of (homotopical) Reedy presheaves on the category $0 \to 1 \to 2$, where the composed arrow $0 \to 2$ is a weak equivalence, resulting in notions of *displayed algebra* and *displayed algebra section* which can be shown to be equivalent to those defined in [6].

It remains to show that categories of algebras have limits. We restrict our attention to finite limits, because they are sufficient for our purposes, and they can be expressed without having to strengthen the requirements on the base model $\mathcal{C}$. We prove the existence of finite limits in categories of algebras in two steps. First, we enhance our original model by adding a type of markings over cones of finite diagrams to the contexts (marked semisimplicial types), preserved by context morphisms; for types, marked cones are required to be limiting (relatively to the context). Then, we refine this model by adding to each type a choice of a marked cone for every diagram.

The first model is a reasonably simple modification of the standard Reedy model of presheaves over a direct category, with some extra work for the type forming operations. The second model is more ad-hoc, but the universe and Π-types in the first model naturally extend to the second. Therefore, we get an interpretation of the theory of signatures into (internal) finitely complete higher categories which is compatible with the previously established interpretation, proving that every higher category of algebras has finite limits.

3 Conclusion and further work

So far, we have shown that every signature for a higher inductive-inductive definition admits a higher category of algebras with finite limits, and that initiality in such categories is equivalent to a syntactically defined induction principle.

However, we have not proved that such initial algebras exist. One could *postulate* the existence of such initial algebras as an axiom of the theory. This could be justified by showing, for example, that in the standard interpretation of homotopy type theory into simplicial sets (within a classical metatheory), the corresponding higher categories of algebras are locally presentable, hence in particular have initial objects.

Alternatively, we speculate that it might be possible to find a general enough *induction principle*, which is similarly justifiable in a classical metatheory, but such that it does not depend on the specifics of the theory of signatures. Then one could assume such principle as an axiom, and prove the existence of initial algebras internally.

References

[1] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In *International conference on foundations of software science and computation structures*, pages 293–310. Springer, 2018.

[2] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending Homotopy Type Theory with strict equality. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL annual conference on Computer Science Logic (CSL 2016)*, volume 62, pages 21:1–21:17, Dagstuhl, Germany, 2016.

[3] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *ArXiv e-prints*, May 2017.

[4] David Ayala and John Francis. Fibrations of ∞-categories. *ArXiv e-prints*, February 2017.

[5] Yonatan Harpaz. Quasi-unital ∞-categories. *Algebraic and Geometric Topology*, 15(4):2303–2381, 2015.

[6] Ambrus Kaposi and András Kovács. A Syntax for Higher Inductive-Inductive Types. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108, pages 20:1–20:18, Dagstuhl, Germany, 2018.

[7] Ambrus Kaposi and András Kovács. Signatures and induction principles for higher inductive-inductive types. *ArXiv e-prints*, February 2019.

[8] Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, pages 1–75, Jan 2015.

Generalizations of Quotient Inductive-Inductive Types *

András Kovács and Ambrus Kaposi

Eötvös Loránd University, Budapest, Hungary
kovacsandras|akaposi@inf.elte.hu

Quotient inductive-inductive types (QIITs) are the most general class of inductive types studied thus far in a set-truncated setting, i.e. in the presence of uniqueness of identity proofs (UIP). In the current work, we develop QIITs further, focusing on applications in practical metatheory of type theories. We extend previous work on QIITs [5] with the following:

1. **Large constructors, large elimination** and algebras at different universe levels. This fills in an important formal gap; large models are routinely used in the metatheory of type theories, but they have not been presented explicitly in previous QIIT literature.

2. **Infinitary constructors**. This covers real, surreal [6] and ordinal numbers. Additionally, the theory which describes QII signatures is itself a large and infinitary QIIT, which allows the theory of signatures to describe its own signature (modulo universe levels), and provide its own model theory. This was not possible previously in [5], where only finitary QIITs were described.

3. **Recursive equations**, i.e. equations appearing as assumptions of constructors. These have occurred previously in syntaxes of cubical type theories, as boundary conditions [4, 1, 2].

4. **Sort equations**. Sort equations were included in Cartmell's generalized algebraic theories (GATs) [3], which overlap significantly with finitary QIITs. Sort equations appear to be useful for algebraic presentations of Russell-style and cumulative universes [7].

Self-describing signatures

In the current work, we would also like to streamline and make more rigorous the specification of signatures. Previous descriptions of GATs [3, 7] used raw syntax with well-formedness relations to describe signatures, which is rather unwieldy to formally handle. Also, the precursor of the current work [5] used an ad-hoc QIIT to describe signatures, which did not have a model theory worked out, and its existence was simply assumed.

In contrast, equipped with large elimination and self-description, we are able to specify signatures and develop a model theory for signatures, without ever using raw syntax or assuming the existence of a particular QIIT. We do the following in order.

1. We specify a notion of model for the theory of signatures (ToS); this is a category with family (CwF) extended with several type formers, allowing to represent a signature as a typing context, with types specifying various constructors.

2. We say that a signature is a context in an *arbitrary* model, i.e. a function with type $(M : \mathsf{ToS}) \to \mathsf{Con}_M$. This can be viewed as a fragment of a Church-encoding; here we

*Both authors author were supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). The second author was supported by by the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme, Project no. ED_18-1-2019-0030 (Application-specific highly reliable IT solutions).

do not care about encoding the whole syntax of ToS, nor the initiality of the syntax, we only need a representation of signatures and the ability to interpret a signature in a ToS-model. For example, the signature for natural number algebras is a function

$$\lambda(M : \mathsf{ToS}).\, (\bullet_M \triangleright_M\ (N : \mathsf{U}_M) \triangleright_M\ (zero : \mathsf{El}_M\, N) \triangleright_M\ (suc : N \Rightarrow_M\ \mathsf{El}_M\, N))$$

which maps every model M to a typing context in M, consisting of the declaration of a sort and two constructors.

3. We give a semantics for signatures, as a particular M : ToS model which interprets each CwF context Γ as a structured category of Γ-algebras. E.g. for the signature of natural numbers, we get a structured category of $\mathbb{N}$-algebras, with $\mathbb{N}$-homomorphisms as morphisms.

4. We give a (large, infinitary) signature for ToS itself, such that interpreting the signature in the semantic model yields a structured category of ToS-algebras. From this, we acquire notions of *recursion* and *induction*, hence we gain the ability to define further constructions by induction on an assumed initial model of the theory of signatures.

In the above construction, everything is appropriately indexed with universe levels (we omit the details), and there is a "bump" of levels at every instance of self-description.

Extending semantics to infinitary constructors and sort equations.

Previously in [5], contexts in ToS were interpreted as CwFs of algebras with extra structure, substitutions as strict morphisms of such CwFs, and types as displayed CwFs. Infinitary constructors force a major change: substitutions must be interpreted as weak CwF morphisms, and types as CwF isofibrations, which are displayed CwFs with an additional lifting structure for isomorphisms. In short, this means that the semantics of infinitary constructors can be only given mutually with a form of invariance under algebra isomorphisms. Recursive equations similarly require this kind of semantics.

However, strict sort equations are not invariant under isomorphisms. For example, if we have an isomorphism in $\mathbf{Set} \times \mathbf{Set}$ between (A, B) and (A', B'), and we also know that $A = B$ strictly, then it is not necessarily the case that $A' = B'$. This means that a strict semantics for sort equations is incompatible with the isofibration semantics for infinitary constructors. Our current solution is to simply keep the troublesome features apart. Hence we have

1. A theory of signatures supporting recursive equations and infinitary constructors, but no sort equations. This ToS can describe itself, and by a term model construction we can reduce all described QIITs to an assumed syntax of the same ToS. This term model construction is also weakened (i.e. it is up to algebra isomorphisms), hence it is significantly more complicated than in [5].

2. A theory of signatures supporting sort equations, but no recursive equations and infinitary constructors. This ToS is infinitary and has no sort equations, so we can give it a model theory as an infinitary QIIT. This ToS supports a stricter semantics which is not invariant under isomorphisms, and we also have a term model construction. Here, the semantics and the term models are straightforward extensions of [5].

References

[1] Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher type theory i: Abstract cubical realizability. *arXiv preprint arXiv:1604.08873*, 2016.

[2] Carlo Angiuli, Kuen-Bang Favonia Hou, and Robert Harper. Cartesian cubical computational type theory: Constructive reasoning with paths and equalities. *Computer Science Logic 2018*, 2018.

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

[4] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint arXiv:1611.02108*, 2016.

[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] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[7] Jonathan Sterling. Algebraic type theory and universe hierarchies. *arXiv preprint arXiv:1902.08848*, 2019.

3 Type-theoretic systems and tools (WG2)

A Sound and Complete Algorithm for Union and Intersection Types in Coq

Claude Stolze
IRIF, Université de Paris
stolze@irif.fr

This paper shortly presents a certified subtyping algorithm for a type system with intersection, union and the universal type U, as it was published in [5], fully certified in Coq [7] and actually used in a prototype of a LF-based theorem prover Bull [6], based on intersection and union types. The soundness and completeness of such an algorithm is not trivial, even though it is a crucial part of the proof of the decidability of type reconstruction and type checking.

Modern theorem provers such as Coq allow us to design and certify algorithms such as the ones for subtyping. We have designed and proved properties for our subtyping algorithm and, thereafter, we certified them in Coq, in the spirit of Bessai's Coq implementation of the subtyping algorithm for intersection types [3]. The full source code of the Coq implementation can be found at `https://github.com/cstolze/Bull-Subtyping`. The certification of the algorithm occurs in two steps: first, we define the subtyping relation and prove some basic properties; then, we implement the subtyping algorithm and show it is sound and complete *w.r.t.* the subtyping relation. The Coq implementation can then be extracted into a valid OCaml (or Haskell) program.

1 The algorithm, shortly explained

The types have the following BNF syntax $\sigma, \tau, \rho ::= \alpha \mid \sigma \cap \sigma \mid \sigma \cup \sigma \mid \sigma \to \sigma \mid \mathtt{U}$. Subtyping is defined as the theory Ξ from [1]. The subtyping algorithm proceeds structurally on some normal form of the types. We thus define the Arrow Normal Form (ANF) as follows :

Definition 1 (ANF)**.** A type is in *Arrow Normal Form* (ANF) if: it is a type variable, or it is a type $\sigma \to \tau$, where σ is an intersection of ANFs (or U) and τ is a union of ANFs. Note that U is not an ANF.

Definition 2 (CANF and DANF)**.** These normal forms are similar to the usual *Conjunctive* and *Disjunctive Normal Forms* (CNF and DNF) found in boolean algebras. An intersection of unions of ANFs is called a *Conjunctive Arrow Normal Form* (CANF), or a union of intersections of ANFs is called a *Disjunctive Arrow Normal Form* (DANF). The type U is considered to be a CANF and a DANF.

We use four rewriting subroutines, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathscr{R}_3$, and $\mathscr{R}_4$, in order to rewrite types in normal form.
The first routine $\mathscr{R}_1$ removes all useless occurrences of U.

Definition 3. (Subroutine $\mathscr{R}_1$)
The term rewriting system $\mathscr{R}_1$ (called `deleteOmega` in the Coq code) is defined as follows: i) $\mathtt{U} \cap \sigma$ and $\sigma \cap \mathtt{U}$ rewrite into σ; ii) $\mathtt{U} \cup \sigma$ and $\sigma \cup \mathtt{U}$ rewrite into U; iii) $\sigma \to \mathtt{U}$ rewrites into U.

The subroutines $\mathscr{R}_2$ and $\mathscr{R}_3$ rewrite a type in conjunctive and disjunctive normal form, respectively.

Definition 4. (Subroutines $\mathscr{R}_2$ and $\mathscr{R}_3$) i) The term rewriting system $\mathscr{R}_2$ rewrites a type into its CNF; ii) The term rewriting system $\mathscr{R}_3$ rewrites a type into its DNF.

Submitted to:
TYPES 2020

Subroutine $\mathscr{R}_4$ rewrites a type as an intersection of ANFs.

Definition 5. (Subroutine $\mathscr{R}_4$) The term rewriting system $\mathscr{R}_4$ rewrites an arrow type into an intersection of ANFs, it is defined as follows: i) $\sigma \to \tau$ rewrites into $\mathscr{R}_3(\sigma) \to \mathscr{R}_2(\tau)$; ii) $\cup_i \sigma_i \to \cap_h \tau_h$ rewrites into $\cap_i(\cap_h(\sigma_i \to \tau_h))$.

We can finally introduce the main algorithm $\mathscr{A}$ as follows:

Definition 6. (Algorithm $\mathscr{A}$)

The main algorithm $\mathscr{A}$ takes as inputs two types σ in DANF and τ in CANF, and decides whether $\sigma \leqslant \tau$ by structural induction as follows:

- if σ and τ are two type variables α and β, then $\sigma \leqslant \tau$ if, and only if, $\alpha \equiv \beta$;
- if $\tau \equiv \mathtt{U}$, then $\sigma \leqslant \tau$;
- if $\sigma \equiv \mathtt{U}$ and $\tau \not\equiv \mathtt{U}$, then $\sigma \not\leqslant \tau$;
- if $\sigma \equiv \sigma_1 \cup \sigma_2$, then $\sigma \leqslant \tau$ if, and only if, $\sigma_1 \leqslant \tau$ and $\sigma_2 \leqslant \tau$;
- if $\tau \equiv \tau_1 \cap \tau_2$, then $\sigma \leqslant \tau$ if, and only if, $\sigma \leqslant \tau_1$ and $\sigma \leqslant \tau_2$;
- if $\sigma \equiv \sigma_1 \cap \sigma_2$, then $\sigma \leqslant \tau$ if, and only if, $\sigma_1 \leqslant \tau$ or $\sigma_2 \leqslant \tau$;
- if $\tau \equiv \tau_1 \cup \tau_2$, then $\sigma \leqslant \tau$ if, and only if, $\sigma \leqslant \tau_1$ or $\sigma \leqslant \tau_2$;
- if $\sigma \equiv \sigma_1 \to \sigma_2$ and $\tau \equiv \tau_1 \to \tau_2$, then $\sigma \leqslant \tau$ if, and only if, $\tau_1 \leqslant \sigma_1$ and $\sigma_2 \leqslant \tau_2$;
- for all other cases, $\sigma \not\leqslant \tau$.

1.1 Soundness and correctness of the algorithm

Theorem 1 (Soundness of $\mathscr{A}$)**.**
Let σ (resp. τ) be in DANF (resp. CANF). If $\mathscr{A}(\sigma,\tau)$, then $\sigma \leqslant \tau$. The proof proceeds by induction.

Theorem 2 (Completeness of $\mathscr{A}$)**.** Let σ (resp. τ) be in DANF (resp. CANF), such that $\sigma \leqslant \tau$. We have that $\mathscr{A}(\sigma,\tau)$. The proof proceeds by mutual induction.

The Coq implementation of $\mathscr{A}$ is called `main_algo` and takes as input two types σ and τ, a proof that σ is in DANF and τ in CANF, and returns either a proof that $\sigma \leqslant \tau$, or a proof that $\sigma \not\leqslant \tau$.

References

[1] Franco Barbanera, Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (1995): *Intersection and union types: syntax and semantics*. Information and Computation 119(2), pp. 202–230.

[2] Yves Bertot & Pierre Castéran (2004): *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[3] Jan Bessai (2016): *Extracting a formally verified Subtyping Algorithm for Intersection Types from Ideals and Filters*. Talk at TYPES, slides.

[4] Alain Frisch, Giuseppe Castagna & Véronique Benzaken (2008): *Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types*. Journal of the ACM 55(4), pp. 19:1–19:64.

[5] Luigi Liquori & Claude Stolze (2017): *A Decidable Subtyping Logic for Intersection and Union Types*. In: Topics In Theoretical Computer Science (TTCS), Lecture Notes in Computer Science 10608, Springer-Verlag, pp. 74–90.

[6] Claude Stolze (2019): *Bull*. `https://github.com/cstolze/Bull`.

[7] Claude Stolze (2019): *Combining union, intersection and dependent types in an explicitly typed lambda-calculus*. Ph.D. thesis, Université Côte d'Azur, Inria.

Formalizing π-calculus in Guarded Cubical Agda

Niccolò Veltri[1] and Andrea Vezzosi[2]

[1] Tallinn University of Technology, Estonia
niccolo@cs.ioc.ee
[2] IT University of Copenhagen, Denmark
avez@itu.dk

The Nakano modality $\triangleright$ [10] is an operator that, when added to a logic or a type system, encodes time at the level of formulae or types. The formula $\triangleright A$ stands for "A holds one time step in the future", similarly the inhabitants of type $\triangleright A$ are the inhabitants of A in the next time step. The Nakano modality comes with a guarded fixpoint combinator $\mathsf{fix}_A : (\triangleright A \to A) \to A$ ensuring the existence of a solution for all guarded recursive equations in any type. Logically, this corresponds to a version of Löb's axiom for the $\triangleright$ modality.

Guarded recursion has been added to Martin-Löf dependent Type Theory in two different ways: using delayed substitutions as in Guarded Dependent Type Theory (gDTT) [2] or using ticks as in Clocked Type Theory (CloTT) [1]. In these settings, the Nakano modality is employed for constructing guarded recursive types, i.e. recursive types in which the recursive variables are guarded by $\triangleright$. These are computed using the fixpoint combinator at type U, which is the universe of small types. For example, the guarded recursive type of infinite streams of natural numbers is obtained as $\mathsf{Str} = \mathsf{fix}_{\mathsf{U}} X.\, \mathbb{N} \times \triangleright X$ and it satisfies the type equivalence $\mathsf{Str} \simeq \mathbb{N} \times \triangleright \mathsf{Str}$. Recursively defined terms of guarded recursive types are causal and productive by construction.

Dependent type theories with guarded recursion have proved themselves suitable for the development of denotational semantics of programming languages, as demonstrated by Paviotti et al's formalization of PCF [11] and Møgelberg and Paviotti's formalization of FPC in gDTT [8]. Here we continue on this line of work by constructing a denotational model of Milner's early π-calculus in a suitable extension of CloTT. Traditionally, the denotational semantics of π-calculus is developed in specific categories of (presheaves over) profunctors [3] or domains [12, 6]. Fundamentally, the semantic domains have to be sufficiently expressive to handle the non-deterministic nature of π-calculus processes. In domain theoretic semantics, for example, this is achieved by employing powerdomains. Synthetic analogues of these constructions are not available in guarded type theories such as gDTT or CloTT, but it can be constructed if we set our development in extensions of these type systems with Higher Inductive Types (HITs), a characterizing feature of Homotopy Type Theory (HoTT).

We work in Ticked Cubical Type Theory (TCTT) [9], an extension of Cubical Type Theory (CTT) [5] with guarded recursion and the ticks from CloTT. CTT is an implementation of a variant of HoTT, giving computational interpretation to its characteristic features: the univalence axiom and HITs. In particular we will define the countable powerset datatype, $\mathsf{P}_\infty\, A$, as a HIT [4], as it will serve as our synthetic powerdomain.

TCTT also has ticks. The Nakano modality is now indexed over the sort of ticks, $\triangleright(\alpha : \mathsf{tick}).A$, and its inhabitants are to be thought of as dependent functions taking in input a tick β and returning an inhabitant of $A[\beta/\alpha]$. So ticks correspond to resources witnessing the passing of time that can be used to access values available only at future times. We write $\triangleright A$ for $\triangleright(\alpha : \mathsf{tick}).A$ when α does not occur free in A. The $\triangleright$ modality is an applicative functor, its unit is called next. Ticks allow to extend the applicative structure to dependent types.

For the specification of the π-calculus syntax, we assume the existence of a countable set of names, i.e., for every natural number n, we assume given a type $\mathsf{Name}\, n$, the set containing the first n names. Each process can perform an output, an input or a silent action. The type

of actions is indexed by two natural numbers, representing the number of free names and the sum of free and bound names, respectively. The input action binds the input name.

$$\frac{ch, v : \mathsf{Name}\, n}{\mathsf{out}\, ch\, v : \mathsf{Act}\, n\, n} \qquad \frac{ch : \mathsf{Name}\, n}{\mathsf{inp}\, ch : \mathsf{Act}\, n\, (n+1)} \qquad \frac{}{\tau : \mathsf{Act}\, n\, n}$$

The π-calculus syntax includes the nil process, prefixing, binary sums, parallel composition, restriction, a matching operator and replication.

$$\frac{}{\mathsf{end} : \mathsf{Pi}\, n} \qquad \frac{a : \mathsf{Act}\, n\, m \quad P : \mathsf{Pi}\, m}{a \cdot P : \mathsf{Pi}\, n} \qquad \frac{P : \mathsf{Pi}\, n \quad Q : \mathsf{Pi}\, n}{P \oplus Q : \mathsf{Pi}\, n} \qquad \frac{P : \mathsf{Pi}\, n \quad Q : \mathsf{Pi}\, n}{P \| Q : \mathsf{Pi}\, n}$$

$$\frac{P : \mathsf{Pi}\, (n+1)}{\nu P : \mathsf{Pi}\, n} \qquad \frac{x, y : \mathsf{Name}\, n \quad P : \mathsf{Pi}\, n}{\mathsf{guard}\, x\, y\, P : \mathsf{Pi}\, n} \qquad \frac{P : \mathsf{Pi}\, n}{!P : \mathsf{Pi}\, n}$$

The processes in $\mathsf{Pi}\, n$ are quotiented by a structural congruence relation $\approx$, which, among other things, characterizes the replication operator in terms of parallel composition: given a process $P : \mathsf{Pi}\, n$, we have $!P \approx P \| !P$. The early operational semantics is inductively defined as a type family $- \, [-] \mapsto - : \mathsf{Pi}\, n \to \mathsf{Label}\, n\, m \to \mathsf{Pi}\, m \to \mathsf{U}$. Following [7], the type $\mathsf{Label}\, n\, m$ of transition labels include a silent action, free and bound outputs, and free and bound inputs.

For the denotational semantic domain, we consider the guarded recursive type

$$\mathsf{Proc} := \mathsf{fix}_{\mathbb{N} \to \mathsf{U}} X.\, \lambda n.\, \mathsf{P}_\infty(\mathsf{Step}\, (\lambda m.\, \triangleright\, \alpha. X\, \alpha\, m)\, n)$$

where $\mathsf{Step}\, Y\, n := \Sigma(m : \mathbb{N}).\, \mathsf{Label}\, n\, m \times Y\, m$. In other words, $\mathsf{Proc}\, n$ is the type satisfying the type equivalence $\mathsf{Proc}\, n \simeq \mathsf{P}_\infty(\Sigma m : \mathbb{N}.\, \mathsf{Label}\, n\, m \times \triangleright \mathsf{Proc}\, m)$. Let Unfold be the right-to-left morphism underlying the latter equivalence. To each syntactic process $P : \mathsf{Pi}\, n$ we associate a semantic process $[\![P]\!] : \mathsf{Proc}\, n$. The interpretation respects the structural congruence relation, that is $P \approx Q$ implies $[\![P]\!] = [\![Q]\!]$. The early operational semantics transitions are modelled using the membership operation: given $P\, [a] \mapsto Q$ with $a : \mathsf{Label}\, n\, m$, we have $(m, a, \mathsf{next}\, [\![Q]\!]) \in \mathsf{Unfold}\, [\![P]\!]$. Nevertheless, Proc is not closed under name substitutions. Therefore, to obtain a sound interpretation of π-calculus, we need to move to the following type:

$$\begin{aligned}\mathsf{PiMod}\, n := \Sigma(P : \Pi(m : \mathbb{N}).(\mathsf{Name}\, n \to \mathsf{Name}\, m) \to \mathsf{Proc}\, m).\\ \Pi(m, m' : \mathbb{N})(f : \mathsf{Name}\, m \to_{\mathsf{inj}} \mathsf{Name}\, m')(\rho : \mathsf{Name}\, n \to \mathsf{Name}\, m).\\ \mathsf{mapProc}\, f\, (P\, m\, \rho) = P\, m'\, (f \circ \rho)\end{aligned}$$

where $A \to_{\mathsf{inj}} B$ is the type of injective maps between A and B, while $\mathsf{mapProc}$ corresponds to the action of the functor Proc on injective renamings.

TCTT provides an extensionality principle for guarded recursive types: strong bisimilarity is equivalent to path equality [9]. For $\mathsf{Proc}\, n$, this says that semantic early bisimilarity is equivalent to path equality. In our work, we also define a syntactic notion of early bisimilarity and early congruence and we prove the denotational semantics fully abstract wrt. it.

We formalized the whole development in our own version of the Agda proof assistant based on TCTT, called Guarded Cubical Agda, an extension of Vezzosi et al's Cubical Agda [14]. An introduction to Guarded Cubical Agda and a detailed description of the formalization can be found in the full paper [13].

Acknowledgments Niccolò Veltri was supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001). Andrea Vezzosi was supported by a research grant (13156) from VILLUM FONDEN.

References

[1] P. Bahr, H. B. Grathwohl, and R. E. Møgelberg. The clocks are ticking: No more delays! In *Proc. of the 32nd Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS 2017*, pages 1–12, 2017.

[2] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In *Proc. of the 19th Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2016*, pages 20–35, 2016.

[3] G. L. Cattani, I. Stark, and G. Winskel. Presheaf models for the pi-calculus. In *Proc. of the 7th Int. Conf. on Category Theory and Computer Science, CTCS 1997*, pages 106–126, 1997.

[4] J. Chapman, T. Uustalu, and N. Veltri. Quotienting the delay monad by weak bisimilarity. *Math. Struct. in Comp. Sci.*, 29(1):67–92, 2019.

[5] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.

[6] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the pi-calculus (extended abstract). In *Proc. of the 11th Ann. IEEE Symp. on Logic in Computer Science, LICS 1996*, pages 43–54, 1996.

[7] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theor. Comput. Sci.*, 114(1):149–171, 1993.

[8] R. E. Møgelberg and M. Paviotti. Denotational semantics of recursive types in synthetic guarded domain theory. *Math. Struct. in Comp. Sci.*, 29(3):465–510, 2019.

[9] R. E. Møgelberg and N. Veltri. Bisimulation as path type for guarded recursive types. *PACMPL*, 3(POPL):4:1–4:29, 2019.

[10] H. Nakano. A modality for recursion. In *Proc. of the 15th Ann. IEEE Symp. on Logic in Computer Science, LICS 2000*, pages 255–266, 2000.

[11] M. Paviotti, R. E. Møgelberg, and L. Birkedal. A model of PCF in guarded type theory. *Electr. Notes Theor. Comput. Sci.*, 319:333–349, 2015.

[12] I. Stark. A fully abstract domain model for the pi-calculus. In *Proc of the 11th Ann. IEEE Symp. on Logic in Computer Science, LICS 1996*, pages 36–42, 1996.

[13] N. Veltri and A. Vezzosi. Formalizing π-calculus in guarded cubical agda. In *Proc. of the 9th ACM SIGPLAN Int. Conf. on Certified Programs and Proofs, CPP 2020*, pages 1–14, 2020.

[14] A. Vezzosi, A. Mörtberg, and A. Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *PACMPL*, 3(ICFP):87:1–87:29, 2019.

Ordinal Notation Systems in Cubical Agda

Fredrik Nordvall Forsberg[1] and Chuangjie Xu[2]

[1] University of Strathclyde, Glasgow, UK
[2] Ludwig-Maximilians-Universität München, Munich, Germany

Abstract

We present ordinal notation systems representing ordinals below ε_0, using recent type-theoretical innovations such as mutual inductive-inductive definitions and higher inductive types. Ordinal arithmetic can be developed for these systems, and they admit a transfinite induction principle. We prove that the notation systems are equivalent, and so we can transport results between them using the univalence principle. All our constructions have been implemented in cubical Agda.

Introduction

Ordinals and ordinal notation systems play an important role in program verification, since they can be used to prove termination of programs — using ordinals to verify that programs terminate was suggested already by Turing [Tur49]. The idea is to assign an ordinal to each input, and then prove that the assigned ordinal decreases for each recursive call [DM79]. Hence the program must terminate by the well-foundedness of the order on ordinals.

If one wants to carry out such proofs in a theorem prover, one must first represent ordinals inside it. This is usually done via some kind of ordinal notation system (however see Blanchette, Popescu and Traytel [BPT14] for well-orders encoded directly in Isabelle/HOL, and Schmitt [Sch17] for an axiomatic method, which is implemented in the KeY program verification system). Typically, ordinals are represented by trees [Der93, DR92]; for instance, binary trees can represent the ordinals below ε_0 as follows: the leaf represents 0, and a tree with subtrees representing ordinals α and β represents the sum $\omega^\alpha + \beta$. However, an ordinal may have multiple such representations. As a result, traditional approaches to ordinal notation systems [Buc91, Sch77, Tak87] usually have to single out a subset of ordinal terms in order to provide unique representations. Instead, we show how modern type-theoretic features in cubical Agda can be used to directly give faithful representations of ordinals below ε_0. More details can be found in our recent paper [NFXG20].

Agda Formalisation. Our development has been fully formalised in Agda, and can be found at `https://doi.org/10.5281/zenodo.3588624`.

An Ordinal Notation System Using Mutual Definitions

The first feature we use is mutual inductive-inductive definitions [NF13]. This allows us to define an ordinal notation system for ordinals below ε_0 based on Cantor normal forms

$$\omega^{\beta_1} + \omega^{\beta_2} + \cdots + \omega^{\beta_n} \qquad \text{with } \beta_1 \geq \beta_2 \geq \cdots \geq \beta_n.$$

By insisting that the ordinal notations β_i are again in Cantor normal form, and given in decreasing order, we can recover a unique representation of ordinals. To achieve this, we define $\mathsf{MutualOrd} : \mathsf{Type}_0$ simultaneously with an order relation $\_<\_ : \mathsf{MutualOrd} \to \mathsf{MutualOrd} \to \mathsf{Type}_0$, and a function $\mathsf{fst} : \mathsf{MutualOrd} \to \mathsf{MutualOrd}$, which extracts the first exponent of an ordinal in Cantor normal form:

```
data MutualOrd where
  𝟎 : MutualOrd
  ω^_+_[_] : (a b : MutualOrd) → a ≥ fst b → MutualOrd

data _<_ where
  <₁ : 𝟎 < ω^ a + b [ r ]
  <₂ : a < c → ω^ a + b [ r ] < ω^ c + d [ s ]
  <₃ : a ≡ c → b < d → ω^ a + b [ r ] < ω^ c + d [ s ]

fst 𝟎 = 𝟎
fst (ω^ a + _ [ _ ]) = a
```

where we write $a \geq b = (a > b) \uplus (a \equiv b)$. Note how all definitions refer to each other. (It is possible to avoid the simultaneous recursive definition of the function fst by defining its graph inductively instead.) An advantage is that there are no intermediate "junk" terms, and that the more precise types often suggest necessary lemmas to prove. This can be seen already when defining basic operations such as ordinal addition and multiplication. To justify that MutualOrd really represents ordinals, we show that it satisfies transfinite induction:

Theorem. *Transfinite induction holds for MutualOrd, i.e. there is a proof*

```
MTI : (P : MutualOrd → Type ℓ) → (∀ x → (∀ y → y < x → P y) → P x) → ∀ x → P x.
```

An Ordinal Notation System Using Higher Inductive Types

We use the feature of higher inductive types [LS19] that has recently been added to Agda under the `--cubical` flag [VMA19] to define a different ordinal notation system for ordinals below ε_0 as a quotient inductive type [ACD+18]:

```
data HITOrd : Type₀ where
  𝟎 : HITOrd
  ω^_⊕_ : HITOrd → HITOrd → HITOrd
  swap : ∀ a b c → ω^ a ⊕ ω^ b ⊕ c ≡ ω^ b ⊕ ω^ a ⊕ c
  trunc : isSet HITOrd
```

Note how the path constructor swap is used to identify multiple representations of the same ordinal — this way, we again recover uniqueness. Because all operations on HITOrd must respect swap, it is not so straightforward to implement ordinal arithmetic on HITOrd directly. However, it is not hard to implement Hessenberg arithmetic [Hes06] — a variant of ordinal arithmetic which is commutative — using cubical Agda's pattern matching.

MutualOrd and HITOrd are Equivalent

Different representations are convenient for different purposes. For instance, the higher inductive type approach is convenient for defining e.g. commutative Hessenberg arithmetic, while the mutual representation is convenient for ordinary ordinal arithmetic, and proving transfinite induction. Using the univalence principle [UFP13], we can transport constructions and properties between the different systems as needed, after proving that they indeed are equivalent:

Theorem. *MutualOrd and HITOrd are equivalent, i.e. there is* `M≃H : MutualOrd ≃ HITOrd`.

The direction of the equivalence from MutualOrd to HITOrd is easy: we simply forget about the order witnesses. The other direction is more interesting, and basically amounts to implementing insertion sort on MutualOrd.

References

[ACD+18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, volume 10803 of *Lecture Notes in Computer Science*, pages 293–310, Heidelberg, Germany, 2018. Springer.

[BPT14] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Cardinals in Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 111–127, Heidelberg, Germany, 2014. Springer.

[Buc91] Wilfried Buchholz. Notation systems for infinitary derivations. *Archive for Mathematical Logic*, 30:227–296, 1991.

[Der93] Nachum Dershowitz. Trees, ordinals and termination. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 243–250, Heidelberg, Germany, 1993. Springer.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DR92] Nachum Dershowitz and Edward M. Reingold. Ordinal arithmetic with list structures. In Anil Nerode and Michael Taitslin, editors, *Logical Foundations of Computer Science*, volume 620 of *Lecture Notes in Computer Science*, pages 117–138, Heidelberg, Germany, 1992. Springer.

[Hes06] Gerhard Hessenberg. *Grundbegriffe der Mengenlehre*, volume 1. Vandenhoeck & Ruprecht, Göttingen, Germany, 1906.

[LS19] Peter Lefanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, pages 1–50, 2019.

[NF13] Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.

[NFXG20] Fredrik Nordvall Forsberg, Chuangjie Xu, and Neil Ghani. Three equivalent ordinal notation systems in Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20)*, pages 172–185. ACM, 2020.

[Sch77] Kurt Schütte. *Proof Theory*. Springer, Heidelberg, Germany, 1977.

[Sch17] Peter H. Schmitt. A mechanizable first-order theory of ordinals. In Renate Schmidt and Cláudia Nalon, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 10501 of *Lecture Notes in Computer Science*, pages 331–346, Heidelberg, Germany, 2017. Springer.

[Tak87] Gaisi Takeuti. *Proof Theory*. North-Holland Publishing Company, Amsterdam, 2 edition, 1987.

[Tur49] Alan Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, UK, 1949. University Mathematical Laboratory.

[UFP13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: a dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):87:1–87:29, 2019.

Universe Polymorphism Expressed as a Rewriting System

Guillaume Genestier[123]

[1] Université Paris-Saclay, ENS Paris-Saclay, Inria, CNRS, LSV
[2] MINES ParisTech, PSL University
[3] This work was supported by the Cost Action EUTypes CA15123

The $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$ for short) 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.

Cousineau and Dowek [3] showed that $\lambda\Pi/\mathcal{R}$ is well-suited to encode a whole class of rich logics: *Functional Pure Type System* (PTS) [2]. To do so, they use a symbol $\texttt{Univ}_s$ for each sort s, which contains the codes of elements of this sort and the associated decoder $\texttt{T}_s$. Then `code` and `prod` reflect the PTS axioms and rules, respectively. For the simply typed λ-calculus, which is the PTS with $\mathcal{S} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$ and $\mathcal{R} = \{(*, *, *)\}$, the encoding is:

```
symbol Univ□ : TYPE.    symbol T□ : Univ□ ⇒ TYPE.    symbol code*□ : Univ□.
symbol Univ* : TYPE.    symbol T* : Univ* ⇒ TYPE.    T□ code*□ ⟶ Univ*.
symbol prod*** : (A : Univ*) ⇒ (T* A ⇒ Univ*) ⇒ Univ*.
T* (prod*** A B) ⟶ (x : T* A) ⇒ T* (B x).
```

In their encoding, every sort has its own symbol, and every rule has its associated product symbol. However, having an infinite number of symbols and rules is not well-suited for practical implementations. Hence, to encode PTS with an infinite number of sorts, Assaf suggested to have a type `Sort` for sorts and a single symbol for products [1]. For *Full Pure Type Systems*[1] this extension is quite direct: `Univ`, `T`, `code` and `prod` are now symbols in the syntax and the meta-arguments of type `Sort` are now real arguments in the syntax. The peculiarity of each PTS is reflected in the encoding of `Sort` and of the functions `axiom` and `rule`.

Let us suppose that all sorts are of the form Set_ℓ with $\ell \in \mathbb{L}$ called a level [2]. It is common to enrich PTS with *Universe Polymorphism* [4], *i.e.* add the possibility for the user to quantify over universe levels, introducing $\forall \ell, \text{Set}_\ell$ among the terms. Indeed, just like we use polymorphism to avoid declaring a type of lists for each type of elements, we do not want to declare a new type for each level. Hence, we want to declare `List` in $\forall \ell, (A : \text{Set}_\ell) \to \text{Set}_\ell$.

To assign a type to $\forall \ell, \text{Set}_\ell$, a new sort Set_ω is introduced, which is not typable, is the type of no sort and over which one cannot quantify. This sort is for internal purposes only, it is not in the syntax of the system we are encoding (even if it is in the syntax of the encoded version of the system). In addition to this new sort, we add to the encoding a new symbol $\forall_{\mathbb{L}}$ which reprents this universal quantification.

```
symbol setOmega : Sort.                    symbol set : 𝕃 ⇒ Sort.
symbol ∀𝕃 : (f : (𝕃⇒Sort)) ⇒ ((l:𝕃) ⇒ Univ (f l)) ⇒ Univ setOmega.
T _ (∀𝕃 f t) ⟶ (l : 𝕃) ⇒ T (f l) (t l).
```

For instance, the encoding of $\forall \ell, \text{Set}_\ell$ is `∀𝕃 (λ l, axiom (set l)) (λ l, code (set l))`. And its decoding (when applying `T setOmega`) is, as expected, `(l:𝕃) -> Univ (set l)`.

Definition 1 (Translation)**.** Given a well-typed term t in a Universe Polymorphic Full Pure Type System, we translate it by: $|x|$=`x` $|\text{Set}_\ell|$=`code` $\|\text{Set}_\ell\|$; $\|\text{Set}_\omega\|$=`setOmega`;

[1] A PTS is called *full* if axioms and rules are total functions, respectively from $\mathcal{S}$ and $\mathcal{S} \times \mathcal{S}$ to $\mathcal{S}$. This definition is more restrictive than the one given in [5], where axioms are not enforced to be total.

[2] We could also, without difficulty, consider several hierarchies sharing the same levels, like Set_ℓ and Prop_ℓ.

$\|\text{Set}_\ell\|$=`set` $|\ell|_{\mathbb{L}}$, if $\ell \neq \omega$; $|(x : A) \to B|$=`prod` $\|s\|$ $\|s'\|$ $|A|$ (λ`x : T` $\|s\|$ $|A|.|B|$);
$|\lambda x^A.t|$=λ(`x : T` $\|s\|$ $|A|$).$|t|$; $|\forall \ell, A|$=$\forall_{\mathbb{L}}$ ($\lambda \ell : \mathbb{L}.\, \|s\|$) ($\lambda \ell : \mathbb{L}.\, |A|$).
Each time it is used, s is the sort of A and s' the one of B.

It can be noted that the translation $|.|_{\mathbb{L}}$ of levels is not given yet. Indeed, with universe polymorphism, universe levels are open terms, hence, convertibility between universe levels is now an issue. Fortunately, it is the last one, since once this issue is overcomed, the encoding has one of the expected properties: we check at least as much as in the original system.

Theorem 2 (Correctness)**.** If the translation function is such that equality of levels implies convertibility of their translations, if $\Gamma \vdash_P t : A$, in a Universe Polymorphic Full Pure Type System P, then $|\Gamma| \vdash_{\lambda\Pi/P} |t|$: `T` $\|s\|$ $|A|$, where s is the sort of A.

Of course, collapsing all levels satisfies the first hypothesis of the theorem. However it is not satisfactory, since it comes down to do an encoding in the inconsistent PTS with only one sort.

We present a correct and complete rewriting system modulo associativity and commutativity (AC), to decide level equality for the PTS where levels are natural numbers and axioms and rules are respectively the functions successor and max [3] [4]. The whole encoding, written in Dedukti, can be found in github.com/Deducteam/Agda2Dedukti, in the files `theory/Agda.dk` and `theory/univ.dk`.

More formally, given the grammar $t, u ::= x \in \mathcal{X} \mid 0 \mid s\,t \mid \max\, t\, u$, every term t in this grammar has a unique normal form denoted $t{\downarrow}$, such that $t{\downarrow} \equiv_{AC} u{\downarrow}$ if and only if for all $\sigma : \mathcal{X} \to \mathbb{N}, [\![t]\!]_\sigma = [\![u]\!]_\sigma$, where the interpretation of 0, s and max are the expected ones.

We must note that having a confluent system is not an issue here, since we desire the unique normal form property only for some specific terms. We obtain this thanks to the guarantee that all variables are of type $\mathbb{L}$.

With our system a normal form is either a variable, or of the form $\text{Max}\ i\ \{j_k + x_k\}_k$ with $x_1, x_2, \ldots$ distinct variables, and $i, j_1, j_2, \ldots$ ground natural numbers such that for all k, $i \geqslant j_k$. It must be noted, that we do not have $+$ in our original grammar, however encoding $s^n(x)$ as $n + x$ avoids to duplicate infinitely rewrite rules, depending on the number of s applied. The first argument is counting iterations, it is why it is restricted to be a ground natural number.

So that they are not confused with levels, a separate type $\mathbb{N}$ of ground natural numbers is introduced[5]. To encode sets, we use symbols modulo AC, since a set is either empty, a singleton of the form $\{i + x\}$, or the union of two sets. The only non-left-linear rule of the encoding eliminates redundancies, ensuring that all variables in the normal forms are distinct.

```
symbol ∅ : LSet.        infix ⊕ : ℕ⇒𝕃⇒LSet.      infix ac ∪ : LSet⇒LSet⇒LSet.
x ∪ ∅ ⟶ x.              (i ⊕ l) ∪ (j ⊕ l) ⟶ max_ℕ i j ⊕ l.
```

The rule stating that $i + \max(t, u) = \max(i + t, i + u)$ must not break the ordering invariant. Hence it has to update the natural number at the head of `Max : ℕ ⇒ LSet ⇒ 𝕃` [6].

```
Max i  (j ⊕ Max k l)        ⟶ Max (max_ℕ i (j +_ℕ k)) (mapPlus j l).
Max i ((j ⊕ Max k l) ∪ tl) ⟶ Max (max_ℕ i (j +_ℕ k)) (mapPlus j l ∪ tl).
```

We can now reflect the syntax we are interested in, using `Max`.

```
0 ⟶ Max 0_ℕ ∅.                  s x ⟶ Max 1_ℕ (1_ℕ ⊕ x).
max x y ⟶ Max 0_ℕ ((0_ℕ ⊕ x) ∪ (0_ℕ ⊕ y)).
```

[3] It is the level hierarchy behind the proof-assistant Agda, which has two families of sorts Prop_ℓ and Set_ℓ.
[4] The impredicative version, behind Coq and Lean, can also be encoded using a similar technique.
[5] Symbols of type $\mathbb{N}$ are in red and indexed with $\mathbb{N}$.
[6] Rules defining `mapPlus` of type $\mathbb{N} \Rightarrow$ `LSet` $\Rightarrow$ `LSet` can easily be inferred and is not detailed.

References

[1] A. Assaf. *A Framework for Defining Computational Higher-Order Logics.* PhD thesis, École polytechnique, 2015.

[2] 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.

[3] D. Cousineau and G. Dowek. *Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo.* TLCA, LNCS 4583:102-117, 2007.

[4] R. Harper and R. Pollack. *Type Checking with Universes.* TCS 89:107-136, 1991.

[5] L. van Benthem Jutting, J. McKinna and R. Pollack. *Checking Algorithms for Pure Type Systems.* Types, LNCS 806:19-61, 1993.

Towards a coinductive mechanisation of Scilla in Agda

Radu A. Ometita

Alexandru Ioan Cuza University, Iasi, Romania
radu.ometita@info.uaic.ro

Introduction. In this paper, we are using the Agda programming language to extend the formalisation of the Scilla smart contract language, introduced in [7].

Smart contracts are self-executing programs with the terms of the agreement between parties written in code. They run on top of blockchains which are immutable ledgers of transactions. The immutability of blockchains makes patching smart contracts impossible, which in turn makes the static verification of smart contracts very desirable.

Scilla structurally separates contract execution and message passing. Every computation is a standalone atomic transition. Transitions are as expressive as a polymorphic lambda calculus (System F), with an added guarantee that all transitions terminate by excluding general recursion from the language. Messages can only be sent at the end of transitions, establishing clear boundaries between contract execution and communication[1].

While transitions themselves always terminate, there is the possibility of having non-terminating behaviour by recursively invoking transitions. Our contribution is in this area, the specification and verification of a possibly infinite trace produced by a finitely branching transition system.

When introducing infinite traces in the formalisation, we need to be able to reason about infinite data and work with possibly infinite proof objects. Agda models infinite proof objects as coinductive data (using copatterns) and all corecursive calls need to satisfy Agda's productivity checker.

Productivity has previously been guaranteed in Coq and Agda by allowing only terms that respect specific syntactic rules (called syntactic guardedness [6]). The syntactic "guardedness" rule states that a data constructor must guard all corecursive calls. Such syntactic checking cannot, however, look within function application (or composition). As a consequence, this makes proofs that mix induction and coinduction challenging to formulate.

Recently a different approach in working with both productivity and termination has been implemented in Agda, using sized types [4, 3]. This approach uses the type system to guarantee productivity (in the case of coinductive definitions) and termination (in case of inductive definitions).

Using the type system for tracking this information means that functions are no longer opaque. We can see how they affect productivity using the type system. Since Agda uses the same mechanism for tracking termination as it does for tracking productivity, we can combine induction and coinduction in a more principled way[2].

The support for sized types makes Agda a primary candidate for our implementation, and the reason it was chosen over Coq. Initial review of the expressivity of sized types has shown its advantages over the syntactic approaches. When mixing induction and coinduction, if the corecursive call is nested within a function, the compiler will not be able to guarantee that the corecursive call is syntactically guarded [5].

Instead of equality, for infinite data, we define a strong bisimilarity relation. Cubical Agda provides further support for working with infinite relations by identifying them with proposi-

[1] Due to the limited amount of space available, some more details about Scilla will be in the presentation.
[2] I will show an example during the presentation.

tional equality. However, we did not investigate Cubical Agda, as there is no information on the way it interacts with the sized types extension.

Contribution. The mechanisation from [7] uses finite traces for the transition system, and it generates the traces using finite schedules (a list of blockchain states and messages).

The output messages from the transitions are always assumed not to affect the state, except for transferring funds from the contract to users, preventing scenarios where contracts can recursively call themselves (which would be possible if output messages could call a contract transition).

Another limitation, probably a consequence of the previous ones is that we cannot model the interaction between multiple contracts. This is not a limitation of the finite definition of traces; however, a coinductive definition may make reasoning about such interactions more elegant than an inductive one.

The finite representation of traces is sufficient for verifying safety properties of the crowd-funding campaign example from [7]. However, there are many instances of contracts where the interaction with the outside world may lead to possibly infinite traces. Any contract that runs forever should be interested also, in the verification of liveness properties (ex: exchanges and wallets).

The first step we took is to change the finite traces from [7] into infinite traces and finish proving the induction principle that [7] uses to verify state invariants for contracts. At this point, there are still some proofs that we need to complete. However, all of these proofs use induction as they are only safety properties of the contract, so we are confident this is not a critical issue.

The difficulties we encountered thus far are related to the way Agda's design patterns relate to coinductive definitions. Understanding the way type inference works for mutual inductive and coinductive definitions, and applying them lead us to concise and simple code, even in the presence of infinite data.

Another hurdle we encountered was the lack of up-to-date information on Agda's sized types support. The size-types extension does not yet have wide adoption, and there are a couple of different ways sized types are used in different sources ([1] vs [2]), leading to contrasting results. Using them as in [1] made our proof of the inductive principle straight-forward.[3].

The original mechanisation is done in Coq, using the syntactic sugar introduced by the 'ssreflect' extension. The syntactic extension makes the code difficult to read for the uninitiated. However, we kept a similar structure and formulated our properties very close to [7].

This is a work in progress, and the presentation will include the current status of the implementation, focused on our way of addressing the above challenges.

Further work The first step we want to take is to improve our transition code by using the recently published [8] big-step semantics. We represent transitions using Agda code written by hand, similarly to the original paper. However, we would like to use Agda to implement the recently specified monadic big-step semantics from [7].

Next, we will update the temporal properties defined in [7] to include potentially infinite traces and use these new definitions in the rest of the development.

A further step will be to develop the initial automata model adding the possibility of interaction between contracts through the sending of messages. Interacting contracts can create interesting infinite traces whose properties, e.g. liveness are to be analysed.

[3]Due to the limited amount of space available, I will only show some examples in the presentation.

References

[1] Andreas Abel. Equational reasoning about formal languages in coalgebraic style. http://www.cse.chalmers.se/~abela/jlamp17.pdf. Accessed: 2020-02-12.

[2] Andreas Abel. Sized types in agda. http://www.cse.chalmers.se/~abela/talkAIM2008Sendai.pdf. Accessed: 2020-02-12.

[3] Andreas Abel. Miniagda: Integrating sized and dependent types. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010*, volume 5 of *EPiC Series*, pages 18–32. EasyChair, 2010.

[4] Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *J. Funct. Program.*, 26:e2, 2016.

[5] Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2005.

[6] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 1993.

[7] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *CoRR*, abs/1801.00687, 2018.

[8] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *PACMPL*, 3(OOPSLA):185:1–185:30, 2019.

4 Proof assistants and technology (WG2)

PRONOM: a theorem prover and countermodel generator for non-normal modal logics*

Tiziano Dalmonte[1], Sara Negri[2], Nicola Olivetti[1], and Gian Luca Pozzato[3]

[1] Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{tiziano.dalmonte,nicola.olivetti}@lis-lab.fr
[2] Dipartimento di Matematica, Universitá di Genova, Italy sara.negri@unige.it
[3] Dipartimento di Informatica, Universitá di Torino, Turin, Italy gianluca.pozzato@unito.it

Abstract

We describe PRONOM, a theorem prover and countermodel generator for non-normal modal logics recently introduced. PRONOM implements some labelled sequent calculi for the basic system **E** and its extensions with axioms M, N, and C based on bi-neighbourhood semantics. The performances of PRONOM, which is implemented in Prolog, are promising.

1 Introduction

Non-Normal Modal Logics (NNML for short) have been studied since the seminal works by C.I. Lewis, Scott, Lemmon, and Chellas (for an introduction see [1]) in the 1960s. They are a generalization of ordinary modal logics that do not satisfy some axioms or rules of minimal normal modal logic **K**. They have gained interest in several areas such as epistemic and deontic reasoning, reasoning about games, and reasoning about "truth in most of the cases". Non-normal modal logics enjoy a simple semantic characterization in terms of Neighbourhood models: these are possible world models where each world is equipped with a set of neighbourhoods, each one being itself a set of worlds; the basic stipulation is that a modal formula $\Box A$ is true at a world w if the set of worlds which make A true belongs to the neighbourhoods of w. A family of logics is obtained by imposing further closure conditions on the set of neighbourhoods.

Here we describe PRONOM, a theorem PROver for NOnnormal Modal logics. PRONOM implements in Prolog the labelled sequent calculi presented in [3], based on *bi-neighbourhood* semantics where each world has associated a set of *pairs* of neighbourhoods, the idea being that the two components of a pair provide independently a positive and negative support for a modal formula. The calculi are modular and provide a decision procedure for the respective logic. Because of the invertibility of the rules, a finite countermodel in the bi-neighbourhood semantics can be directly extracted from a failed derivation. The implementation closely corresponds to the calculi: each rule is encoded by a Prolog clause, and this correspondence ensures in principle both the soundness and completeness of the theorem prover. PRONOM provides both proof search and countermodel generation: it searches for a derivation of an input formula, but in case of failure, it generates a countermodel in the bi-neighbourhood semantics.

2 Non-Normal Modal Logics and Their Sequent Calculi

We present the classical cube of NNMLs, both axiomatically and semantically. Let Atm be a countable set of propositional variables. The language $\mathcal{L}$ contains formulas given by the following grammar: $A ::= p \mid \bot \mid \top \mid A \vee A \mid A \wedge A \mid A \rightarrow A \mid \Box A$, where $p \in$ Atm. The

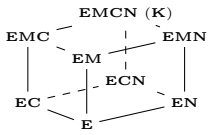
*Supported by the ANR project TICAMORE ANR-16-CE91-0002-01, the Academy of Finland project 1308664 and INdAM project GNCS 2019 "METALLIC #2".

minimal logic **E** in the language $\mathcal{L}$ is defined by adding to classical propositional logic the rule of inference RE $\frac{A \to B \qquad B \to A}{\Box A \to \Box B}$, and can be extended further by choosing any combination of axioms M, C, and N on the left, thus producing the eight distinct logics on the right.

M ▶ $\Box(A \wedge B) \to \Box A$

C ▶ $\Box A \wedge \Box B \to \Box(A \wedge B)$

N ▶ $\Box \top$



We consider a *bi-neighbourhood* semantics for NNMLs [3]. A model is a tuple $\mathcal{M} = \langle \mathcal{W}, \mathcal{N}_b, \mathcal{V} \rangle$, where $\mathcal{W}$ is a non-empty set of worlds (states), $\mathcal{V}$ is a valuation function, and $\mathcal{N}_b$ is a bi-neighbourhood function $\mathcal{W} \longrightarrow \mathcal{P}(\mathcal{P}(\mathcal{W}) \times \mathcal{P}(\mathcal{W}))$, where $\mathcal{P}$ denotes the power set. We say that $\mathcal{M}$ is a *M-model* if $(\alpha, \beta) \in \mathcal{N}_b(w)$ implies $\beta = \emptyset$, it is a *N-model* if for all $w \in \mathcal{W}$ there is $\alpha \subseteq \mathcal{W}$ such that $(\alpha, \emptyset) \in \mathcal{N}_b(w)$, and it is a *C-model* if $(\alpha_1, \beta_1), (\alpha_2, \beta_2) \in \mathcal{N}_b(w)$ implies $(\alpha_1 \cap \alpha_2, \beta_1 \cup \beta_2) \in \mathcal{N}_b(w)$. The forcing relation for boxed formulas is $w \Vdash \Box A$ iff there is $(\alpha, \beta) \in \mathcal{N}_b(w)$ s.t. $\alpha \subseteq [A]$ and $\beta \subseteq [\neg A]$, where $[A]$ is, as usual, the truth set of A in $\mathcal{W}$ obtained by the valuation $\mathcal{V}$.

Let us now recall the labelled calculi for NNMLs based on the bi-neighbourhood semantics, whose language $\mathcal{L}_{\mathsf{LS}}$ extends $\mathcal{L}$ with a set $WL = \{x, y, z, ...\}$ of *world labels*, and a set $NL = \{a, b, c, ...\}$ of *neighbourhood labels*. We define *positive neighbourhood terms*, written $[a_1, ..., a_n]$, as finite multisets of neighbourhood labels, with the unary multiset $[a]$ representing an atomic term. Moreover, if t is a positive term, then $\bar{t}$ is a negative term. Negative terms $\bar{t}$ cannot be proper subterms and cannot be negated. The term τ and its negative counterpart $\overline{\tau}$ are neighbourhood constants. Intuitively, positive (resp. negative) terms represent the intersection (resp. the union) of their constituents, whereas t and $\bar{t}$ are the two members of a pair of neighbourhoods in bi-neighbourhood models. The formulas of $\mathcal{L}_{\mathsf{LS}}$ are of the form: $\phi ::= x : A \mid t \Vdash^{\forall} A \mid t \Vdash^{\exists} A \mid x \in t \mid t \in \mathcal{N}(x)$. Sequents are pairs $\Gamma \Rightarrow \Delta$ of multisets of formulas of $\mathcal{L}_{\mathsf{LS}}$. The fully modular calculi LSE^* are defined by the rules in Figure 1, Section 3 in [3].

3 The Theorem Prover PRONOM

PRONOM [2] is a Prolog implementation of the above calculi. It comprises a set of clauses, each one of them implementing a sequent rule or an axiom of LSE and its extensions. The proof search is provided for free by the mere depth-first search mechanism of Prolog, without any additional ad hoc mechanism. Given a non-normal modal formula F represented by the Prolog term `f`, PRONOM executes the main predicate of the prover, called `prove`, whose only two clauses implement the functioning of PRONOM: the first clause checks whether F is valid and, in case of a failure, the second one computes a model falsifying F. Each clause of the program implements an axiom or rule of the sequent calculi LSE and extensions. To search for a derivation of a sequent $\Gamma \Rightarrow \Delta$, PRONOM proceeds as follows. First of all, if $\Gamma \Rightarrow \Delta$ is an instance of an axiom, the goal will succeed immediately, otherwise the first applicable rule will be chosen, and PRONOM will be recursively invoked on the premises of such a rule. PRONOM proceeds in a similar way for all the rules. The ordering of the clauses is such that the application of the branching rules is postponed as much as possible.

As far as we know, PRONOM is the *first* theorem prover that provides both proof search and countermodel generation for the *whole* cube of non-normal modal logics. Although there are no benchmarks, its performance seems promising. The program PRONOM, as well as all the Prolog source files, including those used for the performance evaluation, are available for free usage and download at `http://193.51.60.97:8000/pronom/`.

References

[1] Brian F. Chellas. *Modal Logic*. Cambridge University Press, 1980.

[2] Tiziano Dalmonte, Sara Negri, Nicola Olivetti, and Gian Luca Pozzato. PRONOM: proof-search and countermodel generation for non-normal modal logics. In Mario Alviano, Gianluigi Greco, and Francesco Scarcello, editors, *AI*IA 2019 - Advances in Artificial Intelligence - XVIIIth International Conference of the Italian Association for Artificial Intelligence, Rende, Italy, November 19-22, 2019, Proceedings*, volume 11946 of *Lecture Notes in Computer Science*, pages 165–179. Springer, 2019.

[3] Tiziano Dalmonte, Nicola Olivetti, and Sara Negri. Non-normal modal logics: Bi-neighbourhood semantics and its labelled calculi. In Guram Bezhanishvili, Giovanna D'Agostino, George Metcalfe, and Thomas Studer, editors, *Advances in Modal Logic 12, proceedings of the 12th conference on Advances in Modal Logic, held in Bern, Switzerland, August 27-31, 2018*, pages 159–178. College Publications, 2018.

[4] Sara Negri. Proof theory for non-normal modal logics: The neighbourhood formalism and basic results. *IfCoLog J. Log. Appl*, 4(4):1241–1286, 2017.

On equality checking for general type theories: Implementation in Andromeda 2*

Andrej Bauer, Philipp G. Haselwarter, and Anja Petković

University of Ljubljana, Ljubljana, Slovenia

Equality checking algorithms are essential components of proof assistants based on type theories [Coq, Agd, dMKA+15, SBF+19, GCST19, AOV17]. They free the user from the burden of proving equalities, and provide computation-by-normalization engines. The type theories found in the most popular type-theoretic proof assistants are carefully designed to have decidable equality. Some systems [Ded, CA16] also allow user extensions to the built-in equality checkers, possibly sacrificing their completeness.

The situation is worse in a proof assistant that supports arbitrary user-definable theories, such as Andromeda 2 [And, BGH+18], where in general no equality checking algorithm may be available. Short of implementing exhaustive proof search, the construction of equality proofs must be delegated to the user (and still checked by the trusted nucleus). While some may appreciate the opportunity to tinker with equality checking procedures, they are surely outnumbered by those who prefer good support that automates equality checking with minimal effort, at least for well-behaved type theories that one encounters in practice.

We have designed and implemented in Andromeda 2 an extensible equality checking algorithm that supports user-defined computation rules (β-rules) and extensionality rules (interderivable with η-rules). The user needs only to provide the equality rules they wish to use, after which the algorithm automatically classifies them either as computation or extensionality rules (and rejects those that are of neither kind), and devises an appropriate notion of weak head-normal form. In the case of well-behaved type theories such as the simply typed lambda calculus or Martin-Löf type theory with η for dependent products, the algorithm behaves like well-known standard equality checkers. In general, it may be incomplete or non-terminating, but it can never be the source of unsoundness because it resides outside of the trusted nucleus.

Our algorithm is a variant of a type-directed equality checking algorithm [SH06, AS12], described in more detail below. It is implemented in around 1300 lines of OCaml code. The algorithm consults the nucleus to build a trusted certificate of every equality it proves and every term normalization it performs. It is easy to experiment with different sets of equality rules, and dynamically switch between them depending on the situation at hand. Our initial experiments are encouraging, although many opportunities for optimization and improvements await.

Type-directed equality checking. The kind of equality checking algorithm that we employ is comprised of several mutually recursive subroutines:

1. *Weak head-normalize a type A:* the user-provided type computation rules are applied to A to give a sequence of equalities $A \equiv A_1 \equiv \cdots \equiv A_n$, until no more rules apply. Then the heads of A_n are normalized recursively (see below) to obtain $A_n \equiv A'_n$, after which the (certified) equality $A \equiv A'_n$ is output.
2. *Weak head-normalize a term $t : A$:* analogously to normalization of types, the user-provided term computation rules are applied to t until no more rules apply.

*This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

3. *Check equality of types $A \equiv B$:* the types A and B are normalized and their normal forms are compared.
4. *Check equality of normalized types $A \equiv B$:* normalized types are compared structurally, i.e., by an application of a suitable congruence rule. Their subexpressions are compared recursively. For example, to prove $\Pi_{(x:C)} D \equiv \Pi_{(x:C')} D'$, we recursively prove $C \equiv C'$ and $x : C \vdash D \equiv D'$.
5. *Check equality of terms $s \equiv t : A$:*
   (a) *type-directed phase:* normalize the type A and based on its normal form apply user-provided extensionality rules, if any, to reduce the equality to subsidiary equalities,
   (b) *normalization phase:* if no extensionality rules apply, normalize s and t and compare their normal forms.
6. *Check equality of normalized terms $s \equiv t : A$:* normalized terms are compared structurally, analogously to comparison of normalized types.

One needs to choose the notions of "computation rule", "extensionality rule" and "normal form" wisely in order to guarantee completeness. In particular, in the type-directed phase the type at which the comparisons are carried out should decrease with respect to a well-founded notion of size, while normalization should be confluent and terminating. These concerns are external to the system, and so the user is allowed to install rules without providing any guarantees of completeness or termination.

Computation and extensionality rules. Term computation rules, type computation rules, and extensionality rules respectively have the forms

$$\frac{P_1 \ \cdots \ P_n}{\vdash u \equiv v : A} \qquad \frac{P_1 \ \cdots \ P_n}{\vdash A \equiv B} \qquad \frac{P_1 \ \cdots \ P_n \quad \vdash x : A \quad \vdash y : A \quad Q_1 \ \cdots \ Q_m}{\vdash x \equiv y : A}$$

In a term computation rule, $P_1, \ldots, P_n$ are *object premises* that introduce term and type meta-variables, while u must be a term symbol applied to subexpressions in which all the meta-variables appear. An extensionality rule, as above, has object premises $P_1, \ldots, P_n$ and subsidiary equality premises $Q_1, \ldots, Q_m$. We require that every meta-variable introduced by the premises appears in A. To tell whether such a rule applies to $s \equiv t : B$, we pattern match B against A, and proceed to work on the instantiated equality subgoals $Q_1, \ldots, Q_m$.

Heads and normal forms. For the algorithm to work correctly, it needs a notion of normal forms that matches the equality rules. We use *weak head-normal forms*: an expression is said to be in normal form if no computation rule applies to it, and its heads are in normal form. Furthermore, when normal forms are compared structurally, their heads are compared structurally in a recursive fashion, while the remaining arguments are compared as ordinary (non-normal) expressions.

The question arises, how to figure out which arguments of a term or a type symbol are the heads. For instance, how can we tell that the third argument of fst above is a head, while pair has no heads? In our implementation the user may specify the heads directly, or let the algorithm read the heads off the computation rules automatically, as follows: if $\mathsf{s}(u_1, \ldots, u_n)$ appears as a left-hand side of a computation rule, then the heads of s are those u_i's that are *not* meta-variables, i.e., matching against them does not automatically succeed, and so further normalization is required. By varying the notion of heads we may control how expressions are normalized. For example, strong normal forms are just weak head-normal forms in which all arguments are declared to be heads.

References

[Agd] The Agda proof assistant. `https://wiki.portal.chalmers.se/agda/`.

[And] The Andromeda proof assistant. `http://www.andromeda-prover.org/`.

[AOV17] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proceedings of the ACM on Programming Languages*, 2(POPL), December 2017.

[AS12] Andreas Abel and Gabriel Scherer. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science*, Volume 8, Issue 1, 2012.

[BGH+18] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Christopher A. Stone. Design and implementation of the Andromeda proof assistant. In *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*, volume 97 of *LIPIcs*, pages 5:1–5:31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[CA16] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In *22nd International Conference on Types for Proofs and Programs TYPES 2016*, University of Novi Sad, 2016.

[Coq] The Coq proof assistant. `https://coq.inria.fr/`.

[Ded] The Dedukti logical framework. `https://deducteam.github.io`.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *25th International Conference on Automated Deduction (CADE 25)*, August 2015.

[GCST19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019.

[SBF+19] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL), December 2019.

[SH06] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 7(4):676–722, 2006.

Proof search for full intuitionistic propositional logic through a coinductive approach for polarized logic

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

We address proof search in full intuitionistic propositional logic, which besides implication also includes at least binary conjunction and disjunction. While the former is regarded as a minor complication which only asks for little extra effort when doing metatheoretic studies, the latter connective presents challenges. It is well-known that in presence of disjunction, also permutative/commuting conversions have to be taken into account to obtain the subformula property. Focused, cut-free sequent calculi provide an appropriate formalization of proofs where such challenges are met. We adopt a complete variant of LJT [6], henceforth called IPL, with some extra simplifications related to eta-expansion, also seen in the recent work by Ferrari and Fiorentini [5, Thm. 2]—but there in the format of natural deduction and without proof terms.

The authors developed for the implicational fragment a coinductive approach to proof search, with which new decision procedures are obtained in recently published work [3]. Our goal is to extend the approach to full intuitionistic logic. However, instead of trying the approach on IPL, we apply it to a more general focused system, into which IPL (and presumably other focused systems) can be embedded: it is a minor variant of cut-free $\lambda^{\pm}_{G}$, a sequent calculus developed by the first author [1] for full *polarized* intuitionistic propositional logic. We will call this minor variant PIPL. The embedding of IPL into PIPL is reminiscent of Girard's embedding of intuitionistic logic into linear logic—but here the target logic is polarized and not linear. The main point, however, is that the image of the embedding enjoys the conservativity property that allows the study of proof search in IPL via PIPL.

The guiding idea of our coinductive approach to proof search is to represent the entire search space of proofs for a given sequent as a single expression. This requires extending the concept of proof expression in two directions: choice points are added to represent the possible application of different proof rules or the different applications of the same proof rule with varying parameters; and, since naive proof search can run into cycles, we adopt a "coinductive representation of proof candidates", that is, we replace the inductive interpretation of proof expressions by a coinductive one, so that they may represent non-wellfounded trees of locally correct applications of proof rules. These expressions are called "forests", and they serve for a precise mathematical specification of our problems of proof search. The algorithmic counterpart that has to be validated against those specifications consists of "finitary forests" that obey an inductive grammar but feature fixed-point expressions that bind meta-variables whose types are sequents (thus finitary forests are not given as a subset of the forests, but they arise syntactically from an inductive definition that also involves explicit fixed points).

In the logic of PIPL, atomic formulas and connectives have one of two polarities, positive or negative. The polarity of a formula is that of its outermost formula constructor. There exist two special formula constructors to invert the polarity (the polarity shifts). The polarity of a connective determines on which side of the sequents the connective is to be focused and inverted. System PIPL is a reworking of Simmons' focused sequent calculus [7] with five categories of sequents, two for the focusing phase (on the left or on the right), two for the inversion phase (idem), and one category of stable sequents $\Gamma \vdash A$—those which require a decision from the "user" of the proof system in order to allow proof search to proceed.

To apply our methodology of coinductive proof search to PIPL, we first define the "forests" of PIPL: based on the five categories of sequents, we have five categories of PIPL proof expressions, and these notions are extended with choice points and read coinductively, as explained before. We call forests those raw coinductive expressions with choice points. While the overall definition is coinductive, only infinite cycling through the category of stable sequents is authorized, and the other simultaneous definitions are by themselves inductive (this can be made precise with stacked least and greatest fixed points over suitable monotone set operators, or, more suggestively, by imposing the parity condition on infinite unfoldings of the non-terminals representing the five categories, where the one corresponding to stable sequents gets priority 2 and the others priority 1). The rationale for this restriction w. r. t. a fully coinductive interpretation is that (1) the other forests would never be typable by sequents, which is why we can safely omit them, and (2) that our corecursive proof constructions when reasoning on forests need to be productive, which would not always be the case for the unrestricted notions.

Each sequent σ can be assigned its search space $\mathcal{S}(\sigma)$ which is a forest of the appropriate category. "Members" of forests in general can be defined inductively, and the definition of $\mathcal{S}(\sigma)$ is adequate in the sense that precisely the inhabitants of σ are the members of the forest $\mathcal{S}(\sigma)$. In our previous work [2, 3], we also considered coinductive members, in order to address questions like "Can proof search for a sequent σ run into loops?", but this is not yet developed for PIPL.

The "finitary forests" of PIPL also come in five categories, and fixed-point variables are only introduced for stable sequents $\Gamma \vdash R$, where R is a "right formula" [1], i. e., positive or a negative atom. There are syntax elements for all proof constructors in the finitary forests, but for stable sequents of form $\sigma := \Gamma \vdash R$ there can also be just a fixed-point variable X of type σ or an expression of the form $\mathsf{gfp}\, X^{\sigma}.E$, with E a representation of all possible applications of the *return* or *coreturn* construction to get a (stable) expression, where the former corresponds to pursuing proof search with focus on a positive formula on the right-hand side and the latter on continuing with focus on a negative formula on the left-hand side. The methodology of our previous publications is applied analogously to specify recursively the outcome of a finitary representation $\mathcal{F}(\sigma)$ of the search space of each sequent σ, by help of a finite list of typed fixed-point variables as an auxiliary parameter. Most notably, for the rules that invert a positive formula in the left-hand side of a sequent and introduce new variables into context Γ, the type of the last fixed-point variable in the list is adapted accordingly. The proof of Lemma 52 in our arXiv paper [2] can be adapted very easily to show that $\mathcal{F}(\sigma)$ is indeed a finitary forest: the main task is to show that the recursion terminates (and this exploits the subformula property).

As in our previous work, a fixed-point variable X can occur in the finitary forest with a different type than at the binding occurrence right after gfp. The semantic interpretation of X needs to cope with varying types and this is by way of a "decontraction" operation as in our previous work: if in a context Γ, a type A is given to more than one variable, say to x and y, one can "expand" a forest T built for x only by adding an alternative with y in place of x, and this independently for all occurrences of x in T. In general, given a forest T, its decontraction $[\Gamma'/\Gamma]T$ is defined when Γ' only has more names for the same assumed formulas as Γ, and our example could be written as $[x : A, y : A/x : A]T$, or even more intuitively as $[(x + y)/x]T$, suggesting a choice between x and y for every occurrence of x.

Future work concerns the treatment of proof efforts with loops, in particular decidability of the question whether they exist, and other questions related to finiteness, including a variant of König's lemma that we described for the implicational fragment in very recent work [4]: the task is to "prune" search spaces in a way that guarantees that they are infinite only if proof search can run into an infinite loop.

References

[1] José Espírito Santo. The Polarized λ-calculus. ENTCS, vol. 332, pp. 149–168, 2017.

[2] José Espírito Santo, Ralph Matthes, and Luís Pinto. A coinductive approach to proof search through typed lambda-calculi. https://arxiv.org/abs/1602.04382, July 2016.

[3] *idem.* Inhabitation in simply-typed lambda-calculus through a lambda-calculus for proof search. *Mathematical Structures in Computer Science*, vol. 29, pp. 1092–1124, September 2019.

[4] *idem.* Decidability of several concepts of finiteness for simple types. *Fundamenta Informaticae*, vol. 170, no. 1–3, pp. 111–138, October 2019.

[5] Mauro Ferrari and Camillo Fiorentini. Goal-Oriented Proof-Search in Natural Deduction for Intuitionistic Propositional Logic. *Journal of Automated Reasoning*, vol. 62, pp. 127–167, 2019.

[6] Hugo Herbelin. Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes. *PhD thesis, Université Paris 7*, 1995.

[7] Robert J. Simmons. Structural Focalization. *ACM TOCL*, vol. 15, issue 3, article 21, 2014.

Modal Induction for Elementary Proofs

Giulio Fellin[1,2], Sara Negri[2,3], and Peter Schuster[1]

[1] Università di Verona, Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona, Italy
{giulio.fellin,peter.schuster}@univr.it
[2] University of Helsinki, Department of Philosophy
P.O. Box 24 (Unioninkatu 40 A), 00014 University of Helsinki, Finland
{giulio.fellin,sara.negri}@helsinki.fi
[3] Università di Genova, Dipartimento di Matematica
via Dodecaneso 35, 16146 Genova, Italy
sara.negri@unige.it

As a rule of thumb, induction principles are more constructive than the corresponding extremum principles. Characteristic examples include Aczel's Set Induction [1,3,4] versus von Neumann and Zermelo's Axiom of Foundation, and Raoult's Open Induction [5,8,19] as opposed to Zorn's Lemma. Since Peano, induction anyway is the main tool for neatly capturing the infinite by representing potentially incomplete processes of generation. This is well reflected by the ubiquity of inductive definitions in today's mathematical logic and theoretical informatics, not least in constructive set and type theories [2,14].

While the usual formulations of induction go beyond first-order logic [9,10], we now use tools of modal logic [6,17] to obtain rather first-order variants of induction. To verify the practicability of these variants in proof practice, we return to the aforementioned examples. Among other things, we prove with the modal companions of induction and with modal logic in sequent-calculus style [16] that every inductive binary relation is irreflexive, and that every meet-closed inductive predicate on a poset propagates from the irreducible elements to any element whatsoever [20]. As a by-product, we gain insights into irreflexivity and transitivity.

Modal logic is obtained from propositional logic by adding the modal operator $\Box$ to the language of propositional logic. A *Kripke model* [12] is a set X together with an *accessibility relation* R, i.e. a binary relation between elements of X, and a *valuation* val, i.e. a function assigning one of the truth values 0 or 1 to an element x of X and an atomic formula P. The usual notation is $x \Vdash P$ for $\mathrm{val}(x,P) = 1$.

We read "xRy" as "y is *accessible* from x" and we read "$x \Vdash P$" as "x *forces* P". Valuations are extended in a unique way to arbitrary formulae by means of inductive clauses:

$$
\begin{aligned}
&x \nVdash \bot \\
&x \Vdash A \supset B \text{ if and only if } x \Vdash A \Rightarrow x \Vdash B \\
&x \Vdash A \wedge B \text{ if and only if } x \Vdash A \text{ and } x \Vdash B \\
&x \Vdash A \vee B \text{ if and only if } x \Vdash A \text{ or } x \Vdash B \\
&x \Vdash \Box A \text{ if and only if } \forall y(xRy \Rightarrow y \Vdash A)
\end{aligned}
$$

We assume that $x \Vdash P$ is decidable for every $x \in X$ and each atomic formula P, which carries over to arbitrary formulae by the inductive clauses. With the intended applications in mind, in place of R we use the inverse accessibility relation $<$, i.e. we stipulate

$$y < x \text{ if and only if } xRy\,.$$

The pair $(X,<)$ is then dubbed *Kripke frame*.

From [15] we adopt the sequent calculus **G3K** for the *basic modal logic* **K**. We consider two new rules on top of **G3K** (rule $R\Box$-*GLI* is rule $R\Box$-*L* of [16]):

$$\frac{y\colon \Box A, \Gamma \to \Delta, y\colon A}{\Gamma \to \Delta, y\colon A}\, NI \qquad \frac{y < x, y\colon \Box A, \Gamma \to \Delta, y\colon A}{\Gamma \to \Delta, x\colon \Box A}\, R\Box\text{-}GLI$$

Both rules come with the variable condition that y does not appear in Γ, Δ.

Proposition 1. *Let $(X, <)$ be a Kripke frame.*

1. *Rule NI is equivalent to $(X, <)$ satisfying* Noetherian Induction*, viz.*

$$\forall y(\forall z < y\, Ez \Rightarrow Ey) \Rightarrow \forall y\, Ey$$

   for all predicates $E(x)$ of the elements x of X.

2. *Rule $R\Box$-GLI is equivalent to $(X, <)$ satisfying* Gödel–Löb Induction*, viz.*

$$\forall x(\forall y < x(\forall z < y\, Ez \Rightarrow Ey) \Rightarrow \forall y < x\, Ey)$$

   for all predicates $E(x)$ of the elements x of X.

This allows us to transform algebraic proofs into modal derivations as follows:

Proof pattern Let X be a set endowed with a binary relation $<$. Suppose that we need to show a statement of the form $\forall y\, E(y)$ by way of Noetherian Induction, or a statement of the form $\forall x \forall y < x\, E(y)$ by way of Gödel–Löb Induction. We consider $(X, <)$ as a Kripke frame, and build a Kripke model by fixing a propositional variable P and defining the valuation

$$\mathrm{val}\colon (x, P) \mapsto \begin{cases} 1 & \text{if } E(x) \\ 0 & \text{otherwise} \end{cases}$$

We usually introduce a pair of rules to express the definition of val. We then proceed as follows:

1. For Noetherian Induction, derive the sequent $y\colon \Box P \to y\colon P$ and apply rule *NI* to obtain $\to y\colon P$, which is indeed equivalent to $\forall y\, E(y)$;
2. For Gödel–Löb Induction, derive the sequent $y < x, y\colon \Box P \to y\colon P$ and apply rule $R\Box$-*GLI* to obtain $\to x\colon \Box P$, which is indeed equivalent to $\forall x \forall y < x\, E(y)$.

Applications With modal sequent calculi [16] we could prove, among other things, that Gödel–Löb Induction is equivalent to Noetherian Induction plus (an appropriate "encoded" version of) transitivity of $<$; that by Noetherian induction every meet-closed predicate on a poset propagates from the irreducible elements to any element whatsoever [20]; and that Noetherian Induction and Gödel–Löb Induction imply the irreflexitivy of $<$ and its variant $\forall y < x(y \neq x)$, respectively.

Future work The calculus **G3K** is classical, but the applications studied up to now have a purely constructive proof in their algebraic counterpart. This makes us confident that we can replace **G3K** by an intuitionistic modal calculus, such as the one presented in [13]. Furthermore, those applications have not yet suggested a general method to find the rules that characterize $x \Vdash P$; whence we will next try to pin down such a general method.

Other principles related to induction are worth a closer look. Apart from the notions of Noetherianity discussed in [10, 18], there is Grzegorczyk induction [11], which is a weaker form of induction compatible with reflexivity. Also the principles of transitivity and irreflexivity deserve further investigation, especially in connection with cut elimination.

References

[1] Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium '77 (Proc. Conf., Wrocław, 1977)*, volume 96 of *Stud. Logic Foundations Math.*, pages 55–66. North-Holland, Amsterdam, 1978.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory: inductive definitions. In *Logic, methodology and philosophy of science, VII (Salzburg, 1983)*, volume 114 of *Stud. Logic Found. Math.*, pages 17–49. North-Holland, Amsterdam, 1986.

[3] Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical report, Institut Mittag–Leffler, 2000. Report No. 40.

[4] Peter Aczel and Michael Rathjen. Constructive set theory. Book draft, 2010. URL: https://www1.maths.leeds.ac.uk/~rathjen/book.pdf.

[5] Ulrich Berger. A computational interpretation of open induction. In F. Titsworth, editor, *Proceedings of the Nineteenth Annual IEEE Symposium on Logic in Computer Science*, pages 326–334. IEEE Computer Society, 2004.

[6] Patrick Blackburn, Maarten Rijke, and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[7] George Boolos and C. Smorynski. *Self-Reference and Modal Logic*, volume 53. Association for Symbolic Logic, 1988. doi:10.2307/2274450.

[8] Thierry Coquand. A note on the open induction principle. Technical report, Göteborg University, 1997. URL: www.cse.chalmers.se/~coquand/open.ps.

[9] Thierry Coquand and Henri Lombardi. A logical approach to abstract algebra. *Math. Structures Comput. Sci.*, 16:885–900, 2006.

[10] Laura Crosilla and Peter Schuster. Finite Methods in Mathematical Practice. In G. Link, editor, *Formalism and Beyond. On the Nature of Mathematical Discourse*, volume 23 of *Logos*, pages 351–410. Walter de Gruyter, Boston and Berlin, 2014.

[11] Roy Dyckhoff and Sara Negri. A cut-free sequent system for grzegorczyk logic, with an application to the gödel–mckinsey–tarski embedding. *J. Logic Computation*, 26:169–187, 2016.

[12] Saul A. Kripke. Semantical analysis of modal logic. I. Normal modal propositional calculi. *Z. Math. Logik Grundlagen Math.*, 9:67–96, 1963.

[13] Paolo Maffezioli, Alberto Naibo, and Sara Negri. The Church–Fitch knowability paradox in the light of structural proof theory. *Synthese*, 190(14):2677–2716, 2013.

[14] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[15] Sara Negri. Proof analysis in modal logic. *J. Philos. Log.*, 34(5–6):507–544, 2005.

[16] Sara Negri and Jan von Plato. *Proof Analysis. A Contribution to Hilbert's Last Problem*. Cambridge University Press, Cambridge, 2011.

[17] Eugenio Orlandelli and Giovanna Corsi. *Corso di logica modale proposizionale*, volume 1169 of *Studi Superiori*. Carrocci editore, 2019.

[18] Hervé Perdry and Peter Schuster. Noetherian orders. *Math. Structures Comput. Sci.*, 21:111–124, 2011.

[19] Jean-Claude Raoult. Proving open properties by induction. *Inform. Process. Lett.*, 29(1):19–23, 1988.

[20] Peter Schuster. Induction in algebra: a first case study. In *2012 27th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia*, pages 581–585. IEEE Computer Society Publications, 2012. *Logical Methods in Computer Science* (3:20) 9 (2013).

5 Types for programming languages (WG3)

Subtype Universes

Harry Maclean and Zhaohui Luo

Royal Holloway, University of London, U.K.
harryjmaclean@gmail.com
zhaohui.luo@hotmail.co.uk

Introduction. Martin-Löf [9] introduced the concept of a universe in order to describe collections of types. The universe Type_0 contains all small types, Type_0 itself is contained in Type_1, and so on. This predicative hierarchy of universes is designed to provide expressive power (we can quantify over collections of types) whilst avoiding paradoxes.

In this paper we explore a new form of universe: a collection of subtypes. For any type A we define a universe $U(A)$ as the collection of all *subtypes* of A. We work in a system UTT[$\mathbb{C}$] which is an extension of UTT [6] with coercive subtyping [8] specified by means of a set $\mathbb{C}$ of coherent coercions. Our subtype universe is defined by the addition of two rules (for brevity we use universe formulation *à la Russell*):

$$\frac{\Gamma \vdash H : \mathbf{Type}}{\Gamma \vdash U(H) : \mathbf{Type}}\ U\text{-formation} \qquad \frac{\Gamma \vdash A \leq H : \mathbf{Type}}{\Gamma \vdash A : U(H)}\ U\text{-introduction}$$

U-formation declares that for type H, $U(H)$ is also a type, and U-introduction says that for any type A which is a subtype of H, A is in the universe $U(H)$. Note that H is always contained in $U(H)$ and, as a degenerate case, $U(H)$ contains only H when $\mathbb{C}$ is empty. We now have a type for "all subtypes of H", and can therefore quantify over this type. For example, $\forall(X : U(H)).P(X)$ is the proposition that a particular property P holds for all subtypes of H.

This subtype universe neatly models bounded quantification (of the form $\Pi(A \leq H).P(A)$) whilst avoiding the type checking issues traditionally associated with it [10]. We show that a specific form of this construction is a logically consistent extension of UTT[$\mathbb{C}$], explore applications in programming and natural language semantics, and discuss possible further work.

Application to programming. Subtype universes provide an alternative model for bounded quantification [3]. For example, we can define a polymorphic identity function over all subtypes of H as $id_H = \lambda X.\lambda x.x : \Pi(X : U(H)).X \to X$. In a system without subtype universes or bounded quantification the equivalent function would be $id_H = \lambda x.x : H \to H$. Given an object $h : H'$ where $H' < H$, the expression $id_H(h)$ is well-typed via subtype subsumption, but the result will be an object of type H - we have lost the information that h is of type H', simply by passing it through the identity function. By specifying the argument type and not relying on subsumption, both bounded quantification and subtype universes avoid this problem.

As a more compelling example, consider a type of (non-dependent) records, representing heterogeneous sets of labelled values. We write a record type as $\{x : A, y : B, ...\}$ where x, y are field labels and A, B are field types. We define a subtyping relation on records as follows:

$$\frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma \vdash R : RType}{\Gamma \vdash \{R, x : A\} \leq R : \mathbf{Type}}\ (x \notin R)$$

where R is the type of records and $\{R, x : A\}$ denotes the extension of R by a field x of type A, under the assumption that R does not have a field labelled x. $RType$ is the kind of record types, as described in [7]. We can write the type of a function that extracts the value named "length" as

$$getLength : \Pi(R : U(\{length : Int\})).R \to Int$$

where $U(\{length : Int\})$ is the universe of subtypes of the record type $\{length : Int\}$. $getLength$ can be applied to any record which has a field labelled "length" of type Int.

Unlike bounded quantification, subtype universes can appear anywhere in a type. For example the codomain of the function $f : U(H) \to U(H) \to U(H)$ is a universe, and $U(H) \times A$ is a product type containing a universe.

Application to natural language semantics. In formal semantics based on type theories [11, 5, 4], common nouns are interpreted as types and subtype universes are useful in semantic constructions. For example, they can be employed to model gradable adjectives (words such as "tall"). Gradable adjectives map their arguments on to a totally ordered set of degrees, or scale. For "tall", this scale is height. Consider a universe T of base types containing objects for which we can determine their height, which may contain $Human : T$ and $Building : T$. We can define the type of "tall" as $\text{tall} : \Pi(H : T).\Pi(A : U(H)).(A \to Prop)$.

In words, "tall" is a predicate on subtypes of types in T, and $U(H)$ is the subtype universe for H. Thus if we have a type $Man < Human$ and an object $socrates : Man$ then $\text{tall}(Human, Man, socrates) : Prop$. An example definition is $\text{tall}(H, A, x) \equiv height(x) \geq \xi(H, A)$, where $\xi : \Pi(H : T).(U(H) \to \mathbb{N})$ is a function that calculates the "threshold" height for any subtype of a type in T.

Metatheoretic correctness. We have proved that the extension of UTT[$\mathbb{C}$] with the two universe formation rules is not problematic in a simplified case of the more general system. We pick a specific (but arbitrary) small type H and a single subtyping rule:

$$\frac{\Gamma, x : A \vdash P : Prop}{\Gamma \vdash \Sigma(x : A).P \leq_{\pi_1} A : \mathbf{Type}}$$

where $\pi_1 : \Sigma(x : A).P \to A$ is the first projection for pairs. We prove that all theorems of this simplified system (written $\text{UTT}_H[\mathbb{C}]$) are derivable in UTT[$\mathbb{C}$] via a transformation δ, which maps $U(H)$ to Type_0 but leaves terms otherwise unchanged. A corollary to this is that $\text{UTT}_H[\mathbb{C}]$ is logically consistent, as UTT[$\mathbb{C}$] is a conservative extension of UTT, which is itself a consistent system.

Recent work has extended this result to an arbitrary but coherent set of coercions obeying a restriction that we define here informally: for any coercion $c : A \to B$ we require $A : \text{Type}_i$, $B : \text{Type}_j$ and $i \leq j$. Care must be taken to ensure that coercions cannot themselves mention subtype universes, as otherwise we can derive potentially paradoxical judgements. For example, from the coercion $c : U(H) \to H$ we can derive a judgement $\Gamma \vdash U(H) : U(H)$. The relationship between subtype universes and the existing predicative universe hierarchy is ambiguous and in need of further investigation.

Conclusion. Subtype universes provide a powerful mechanism for quantifying over collections of subtypes without the undecidability issues of bounded quantification. We give several example applications in programming and natural language semantics; there are surely others.

Subtype universes bear similarities to Cardelli's power type [2] $\text{Power}(A)$, a type containing all subtypes of A. Cardelli's formulation uses structural subtyping and a system with the logically inconsistent $\text{Type} : \text{Type}$, whereas our system is built on the logically consistent UTT. Aspinall's λ_{Power} [1] is a predicative and simplified alternative to Cardelli's system, but it has been difficult to prove some of its metatheoretic properties (such as subject reduction).

So far we have focused on a specific (but arbitrary) type $H : \text{Type}_0$. We are hopeful that the result can be extended to any type, and this will be the focus of future work.

References

[1] David Aspinall. Subtyping with power types. In *International Workshop on Computer Science Logic*, pages 156–171. Springer, 2000.

[2] Luca Cardelli. Structural subtyping and the notion of power type. *POPL*, 1988.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[4] S. Chatzikyriakidis and Z. Luo. *Formal Semantics in Modern Type Theories*. Wiley & ISTE Science Publishing Ltd., 2020. (to appear).

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

[6] Zhaohui Luo. *Computation and reasoning*, volume 20. Oxford University Press, 1994.

[7] Zhaohui Luo. Dependent record types revisited. In *Proceedings of the 1st Workshop on Modules and Libraries for Proof Assistants*, pages 30–37. ACM, 2009.

[8] Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: theory and implementation. *Information and Computation*, 223, 2013.

[9] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis Naples, 1984.

[10] Benjamin C Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1), 1994.

[11] A. Ranta. *Type-Theoretical Grammar*. Oxford University Press, 1994.

Flexible Coinduction in Agda

Luca Ciccone[1], Francesco Dagnino[2], and Elena Zucca[2]

[1] University of Torino, Italy
[2] University of Genova, Italy

Inference systems [2, 9] are a widely-used formalism to define predicates. An inference system $\mathcal{I}$ is a set of *rules* $\frac{Pr}{j}$, with $Pr \subseteq \mathcal{U}, j \in \mathcal{U}$, for $\mathcal{U}$ a universe of *judgments*. In a *proof tree* for j, nodes are (labeled with) judgments in $\mathcal{U}$, j is the root, and a node j has children Pr only if there is a rule $\frac{Pr}{j}$ in $\mathcal{I}$. The *inductive interpretation* $Ind(\mathcal{I})$, and the *coinductive interpretation* $CoInd(\mathcal{I})$, are, in proof-theoretic terms, the sets of judgments with, respectively, a finite[1], and a possibly infinite proof tree.

Inference systems with corules [3, 7] are a recently proposed generalization allowing *flexible coinduction*, that is, to define predicates which are neither inductive, nor purely coinductive. An inference system with corules is a pair $(\mathcal{I}, \mathcal{I}_{co})$ where $\mathcal{I}$ and $\mathcal{I}_{co}$ are inference systems, whose elements are called *rules* and *corules*, respectively. Its *interpretation* $FCoInd(\mathcal{I}, \mathcal{I}_{co})$ is the set of judgments with a possibly infinite proof tree in $\mathcal{I}$, where all nodes have a finite proof tree in $\mathcal{I} \cup \mathcal{I}_{co}$, that is, the (standard) inference system consisting of rules and corules. For instance, the following inference system defines the maximal element of a list, where $[x]$ is the list consisting of only x, and $x : u$ the list with head x and tail u.

$$\texttt{(max-h)}\ \frac{}{maxElem\ (x, [x])} \qquad \texttt{(max-t)}\ \frac{maxElem\ (y, u)}{maxElem\ (z, x{:}u)}\, z = \max(x, y)$$

The inductive interpretation gives the correct result only on finite lists. However, the coinductive one fails to be a function. For instance, set L the infinite list $1 : 2 : 1 : 2 : 1 : 2 : \ldots$, any judgment $maxElem\ (L, x)$ with $x \geq 2$ has an infinite proof tree. By adding a corule (in this case a coaxiom), wrong results are "filtered out":

$$\texttt{(max-h)}\ \frac{}{maxElem\ (x, [x])} \qquad \texttt{(max-t)}\ \frac{maxElem\ (y, u)}{maxElem\ (z, x{:}u)}\, z = \max(x, y) \qquad \texttt{(co-max-h)}\ \frac{}{maxElem\ (x, x{:}u)}$$

Indeed, only for $maxElem\ (1{:}2{:}L, 2)$ each node of the infinite proof tree has a finite proof tree in the inference system extended by the corule. We refer to [3, 4, 5, 7] for other examples.

Bounded coinduction [3, 7] is the proof technique, generalizing standard coinduction, to show completeness of an inference system with corules with respect to a specification.

The aim of this work is to investigate how to express in Agda inference systems with corules, and the related proof techniques. To this end, we have to face two challenges.

- Inductive and coinductive predicates can be directly translated into an Agda type, which has in turn an inductive or coinductive semantics. Notably, the former are expressed as inductive types by the `data` construct, whereas, for the latter, the standard library provides a representation based on *sized types* [1, 8] and *thunks*, which are, roughly speaking, suspended computations used to simulate laziness. For predicates which are neither inductive nor purely coinductive, instead, Agda has no built-in support.
- Agda inductive and coinductive types implicitly provide their corresponding induction and coinduction principles. The bounded coinduction principle, instead, needs to be explicitly expressed and proved.

[1] Assuming that sets of premises are finite, otherwise we should say a tree with no infinite paths.

In [6], we have provided a methodology to face the above issues, illustrated by several examples. To express the predicate, we use two Agda types.

- A coinductive type that represents the interpretation of the inference system with corules, that is, $FCoInd(\mathcal{I}, \mathcal{I}_{co})$.

```
data _maxElem_ : Nat → Colist Nat ∞ → Size → Set where
  max-h : ∀ {x xs i} → Thunk.force xs ≡ [] → (x maxElem (x :: xs)) i
  max-t : ∀ {n x z xs i} → Thunk (n maxElem (Thunk.force xs)) i →
           z ≡ max n x → z maxElem-ind (x :: xs) → (z maxElem (x :: xs)) i
```

  Accordingly with the definition, this type internally uses the following other type:

- An inductive type that represents the inductive interpretation of the inference system consisting of rules and corules, that is, $Ind(\mathcal{I} \cup \mathcal{I}_{co})$.

```
data _maxElem-ind_  : Nat → Colist Nat ∞ → Set where
  max-h-ind : ∀ {x xs} → Thunk.force xs ≡ [] → x maxElem-ind (x :: xs)
  max-t-ind : ∀ {n x z xs} → n maxElem-ind (Thunk.force xs) →
                z ≡ max n x → z maxElem-ind (x :: xs)
  co-max-h : ∀ {x xs} → x maxElem-ind (x :: xs)
```

In these types, type constructors nicely correspond to the (meta-)rules and (in the latter) also meta-corules. Moreover, an auxiliary type (called `Step`) is defined, expressing the parametric predicate that, for a given specification, a judgment is the consequence of a rule whose premises satisfy the specification. Finally, the bounded coinduction principle corresponding to the predicate can be proved, using the type `Step` defined before to express consistency.

Among the examples in [6], it is worthwhile to mention that we investigated temporal operators. Generally, we reason inductively to prove a liveness property, and coinductively to prove a safety property. For more complex properties mixing safety and liveness, inference systems with corules look like a promising approach, as we illustrate on the *infinitely-often* example.

In ongoing work we aim at transforming this methodology in an automatic support for a user. We are considering two different approaches.

- To formalize in Agda the meta-theory. That is, to represent a generic inference system with corules as an Agda type parametric on the universe:

```
IS : Set → Set₁
IS U = List U → U → Set
```

  and to develop on top of this significant definitions and theorems, e.g., interpretation and bounded induction. In this way, to express a specific inference system with corules, a user should just supply the arguments. The drawback of this approach from the user's point of view is that the specific (meta-)rules would be not directly "visible" in Agda code.

- To allow a user to write an inference system with corules in a natural syntax, and automatically generate the corresponding Agda types. This could be achieved by an external tool, that is, user definitions could be given to a parser producing the Agda code. An alternative, more interesting and challenging, solution, is to write the inference system as an Agda type, and then use *reflection*, which was recently added in Agda[2].

The two approaches could be combined by automatically generating, in the latter, the arguments to be provide to the parametric Agda type in the former.

[2] https://agda.readthedocs.io/en/v2.6.0.1/language/reflection.html

References

[1] Andreas Abel. MiniAgda: Integrating sized and dependent types. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010*, pages 14–28, 2010. URL: https://doi.org/10.4204/EPTCS.43.2, doi:10.4204/EPTCS.43.2.

[2] Peter Aczel. An introduction to inductive definitions. In *Handbook of Mathematical logic*. North Holland, 1977.

[3] Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.

[4] Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *PACMPL*, 1(OOPSLA):81:1–81:26, 2017.

[5] Davide Ancona, Francesco Dagnino, and Elena Zucca. Modeling infinite behaviour by corules. In *ECOOP'18 - Object-Oriented Programming*, pages 21:1–21:31, 2018.

[6] Luca Ciccone. Flexible coinduction in agda, 2020. URL: https://arxiv.org/abs/2002.06047, arXiv:2002.06047.

[7] Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Logical Methods in Computer Science*, 15(1), 2019. URL: https://doi.org/10.23638/LMCS-15(1:26)2019, doi:10.23638/LMCS-15(1:26)2019.

[8] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 410–423. ACM Press, 1996. URL: https://doi.org/10.1145/237721.240882, doi:10.1145/237721.240882.

[9] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

Soundness conditions for big-step semantics
(Extended Abstract)

Francesco Dagnino[1], Viviana Bono[2], Elena Zucca[1], and Mariangiola Dezani-Ciancaglini[2]

[1] DIBRIS, University of Genova
[2] Computer Science Department, University of Torino

The semantics of programming languages or software systems specifies, for each program/system configuration, its final result, if any. In the case of non-existence of a final result, there are two possibilities:

- either the computation stops with no final result, and there is no means to compute further: *stuck computation*,
- or the computation never stops: *non-termination*.

There are two main styles to define operationally a semantic relation: the *small-step* style [4, 5], on top of a reduction relation representing single computation steps, or directly by a set of rules as in the *big-step* style [2].

Within a small-step semantics it is straightforward to make the distinction between stuck and non-terminating computations, while a typical drawback of the big-step style is that they are not distinguished (no judgement is derived in both cases).

For this reason, even though big-step semantics is generally more abstract, and sometimes more intuitive to design and therefore to debug and extend, in the literature much more effort has been devoted to study the meta-theory of small-step semantics, providing properties, and related proof techniques. Notably, the *soundness* of a type system (typing prevents stuck computation) can be proved by *progress* and *subject reduction* (also called *type preservation*) [6].

Our quest is then to provide a general proof technique to prove the soundness of a predicate with respect to an arbitrary big-step semantics. How can we achieve this result, given that in big-step formulation soundness cannot even be *expressed*, since non-termination is modelled as the absence of a final result exactly like stuck computation? The key idea is the following:

1. We define constructions *yielding an extended version of a given arbitrary big-step semantics*, where the difference between stuckness and non-termination is made explicit. In a sense, these constructions show that the distinction was "hidden" in the original semantics.
2. We provide a general proof technique by identifying *three sufficient conditions* on the original big-step rules to prove soundness.

Keypoint (2)'s three sufficient conditions are *local preservation*, $\exists$*-progress*, and $\forall$*-progress*. For *proving* the result that the three conditions actually ensure soundness, the setting up of the extended semantics from the given one is necessary, since otherwise, as said above, we could not even express the property.

However, the three conditions deal only with the original rules of the given big-step semantics. This means that, practically, in order to use the technique there is no need to deal with the extended semantics. This implies, in particular, that our approach does *not* increase the original number of rules. Moreover, the sufficient conditions are conditions on *single rules*, which makes explicit the proof fragments typically needed in a proof of soundness. Even though this is not

exploited in the ESOP paper [1], this form of *locality* means *modularity*, in the sense that adding a new rule implies adding the corresponding proof fragment only.

As an important by-product, in order to formally define and prove correct the keypoints (1) and (2), we propose a formalisation of "what is a big-step semantics" which captures its essential features. Moreover, we support our approach by presenting several examples, demonstrating that: on the one hand, their soundness proof can be easily rephrased in terms of our technique, that is, by directly reasoning on big-step rules; on the other hand, our technique is essential when the property to be checked (for instance, soundness of a type system) is *not preserved* by intermediate computation steps, whereas it holds for the final result. On a side note, our examples concern type systems, but the meta-theory we present holds for any predicate.

We describe now in more detail the constructions of keypoint (1). Starting from an arbitrary big-step judgment $c \Rightarrow r$ that evaluates *configurations* c into *results* r, the *first construction* produces an enriched judgement $c \Rightarrow_{\mathsf{tr}} t$ where t is a *trace*, that is, the (finite or infinite) sequence of all the (sub)configurations encountered during the evaluation. In this way, by interpreting coinductively the rules of the extended semantics, an infinite trace models divergence (whereas no result corresponds to stuck computation). The *second construction* is in a sense dual. It is the *algorithmic* version of the well-known technique presented in Exercise 3.5.16 from the book [3] of adding a special result wrong explicitly modelling stuck computations (whereas no result corresponds to divergence). Note that the latter approach is still inductive, hence computable, since non-termination is modelled implicitly, differently from the former. However, either approach is useful since it allows reasoning on different properties: notably, by trace semantics and wrong semantics we can express two flavours of soundness, *soundness-may* and *soundness-must*, respectively, and show the correctness of the corresponding proof technique. This achieves our original aim, and it should be noted that *we define soundness with respect to a big-step semantics within a big-step formulation*, without resorting to a small-step style (indeed, the two extended semantics are themselves big-step).

Lastly, we consider the issue of justifying on a formal basis that the two constructions are correct with respect to their expected meaning. For instance, for the wrong semantics we would like to be sure that *all* the cases are covered. To this end, we define a *third construction*, dubbed PEV for "partial evaluation", which makes explicit the *computations* of a big-step semantics, intended as the sequences of execution steps of the naturally associated evaluation algorithm. Formally, we obtain a reduction relation on approximated proof trees, so termination, non-termination and stuckness can be defined as usual. Then, the correctness of traces and wrong constructions is proved by showing they are equivalent to PEV for diverging and stuck computations, respectively.

This is the extended abstract of a paper to appear in ESOP'20 [1].

References

[1] Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In *ESOP 2020 - European Symposium on Programming*, 2020. To appear.

[2] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS'87 - Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Berlin, 1987. Springer.

[3] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, Massachusetts, 2002.

[4] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[5] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[6] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Ordinal Notation Systems in Cubical Agda

Fredrik Nordvall Forsberg[1] and Chuangjie Xu[2]

[1] University of Strathclyde, Glasgow, UK
[2] Ludwig-Maximilians-Universität München, Munich, Germany

Abstract

We present ordinal notation systems representing ordinals below ε_0, using recent type-theoretical innovations such as mutual inductive-inductive definitions and higher inductive types. Ordinal arithmetic can be developed for these systems, and they admit a transfinite induction principle. We prove that the notation systems are equivalent, and so we can transport results between them using the univalence principle. All our constructions have been implemented in cubical Agda.

Introduction

Ordinals and ordinal notation systems play an important role in program verification, since they can be used to prove termination of programs — using ordinals to verify that programs terminate was suggested already by Turing [Tur49]. The idea is to assign an ordinal to each input, and then prove that the assigned ordinal decreases for each recursive call [DM79]. Hence the program must terminate by the well-foundedness of the order on ordinals.

If one wants to carry out such proofs in a theorem prover, one must first represent ordinals inside it. This is usually done via some kind of ordinal notation system (however see Blanchette, Popescu and Traytel [BPT14] for well-orders encoded directly in Isabelle/HOL, and Schmitt [Sch17] for an axiomatic method, which is implemented in the KeY program verification system). Typically, ordinals are represented by trees [Der93, DR92]; for instance, binary trees can represent the ordinals below ε_0 as follows: the leaf represents 0, and a tree with subtrees representing ordinals α and β represents the sum $\omega^\alpha + \beta$. However, an ordinal may have multiple such representations. As a result, traditional approaches to ordinal notation systems [Buc91, Sch77, Tak87] usually have to single out a subset of ordinal terms in order to provide unique representations. Instead, we show how modern type-theoretic features in cubical Agda can be used to directly give faithful representations of ordinals below ε_0. More details can be found in our recent paper [NFXG20].

Agda Formalisation. Our development has been fully formalised in Agda, and can be found at `https://doi.org/10.5281/zenodo.3588624`.

An Ordinal Notation System Using Mutual Definitions

The first feature we use is mutual inductive-inductive definitions [NF13]. This allows us to define an ordinal notation system for ordinals below ε_0 based on Cantor normal forms

$$\omega^{\beta_1} + \omega^{\beta_2} + \cdots + \omega^{\beta_n} \qquad \text{with } \beta_1 \geq \beta_2 \geq \cdots \geq \beta_n.$$

By insisting that the ordinal notations β_i are again in Cantor normal form, and given in decreasing order, we can recover a unique representation of ordinals. To achieve this, we define $\mathsf{MutualOrd} : \mathsf{Type}_0$ simultaneously with an order relation $\_<\_ : \mathsf{MutualOrd} \to \mathsf{MutualOrd} \to \mathsf{Type}_0$, and a function $\mathsf{fst} : \mathsf{MutualOrd} \to \mathsf{MutualOrd}$, which extracts the first exponent of an ordinal in Cantor normal form:

```
data MutualOrd where
  𝟎 : MutualOrd
  ω^_+_[_] : (a b : MutualOrd) → a ≥ fst b → MutualOrd

data _<_ where
  <₁ : 𝟎 < ω^ a + b [ r ]
  <₂ : a < c → ω^ a + b [ r ] < ω^ c + d [ s ]
  <₃ : a ≡ c → b < d → ω^ a + b [ r ] < ω^ c + d [ s ]

fst 𝟎 = 𝟎
fst (ω^ a + _ [ _ ]) = a
```

where we write $a \geq b = (a > b) \uplus (a \equiv b)$. Note how all definitions refer to each other. (It is possible to avoid the simultaneous recursive definition of the function fst by defining its graph inductively instead.) An advantage is that there are no intermediate "junk" terms, and that the more precise types often suggest necessary lemmas to prove. This can be seen already when defining basic operations such as ordinal addition and multiplication. To justify that MutualOrd really represents ordinals, we show that it satisfies transfinite induction:

Theorem. *Transfinite induction holds for MutualOrd, i.e. there is a proof*

```
MTI : (P : MutualOrd → Type ℓ) → (∀ x → (∀ y → y < x → P y) → P x) → ∀ x → P x.
```

An Ordinal Notation System Using Higher Inductive Types

We use the feature of higher inductive types [LS19] that has recently been added to Agda under the `--cubical` flag [VMA19] to define a different ordinal notation system for ordinals below ε_0 as a quotient inductive type [ACD+18]:

```
data HITOrd : Type₀ where
  𝟎 : HITOrd
  ω^_⊕_ : HITOrd → HITOrd → HITOrd
  swap : ∀ a b c → ω^ a ⊕ ω^ b ⊕ c ≡ ω^ b ⊕ ω^ a ⊕ c
  trunc : isSet HITOrd
```

Note how the path constructor swap is used to identify multiple representations of the same ordinal — this way, we again recover uniqueness. Because all operations on HITOrd must respect swap, it is not so straightforward to implement ordinal arithmetic on HITOrd directly. However, it is not hard to implement Hessenberg arithmetic [Hes06] — a variant of ordinal arithmetic which is commutative — using cubical Agda's pattern matching.

MutualOrd and HITOrd are Equivalent

Different representations are convenient for different purposes. For instance, the higher inductive type approach is convenient for defining e.g. commutative Hessenberg arithmetic, while the mutual representation is convenient for ordinary ordinal arithmetic, and proving transfinite induction. Using the univalence principle [UFP13], we can transport constructions and properties between the different systems as needed, after proving that they indeed are equivalent:

Theorem. *MutualOrd and HITOrd are equivalent, i.e. there is* `M≃H : MutualOrd ≃ HITOrd`.

The direction of the equivalence from MutualOrd to HITOrd is easy: we simply forget about the order witnesses. The other direction is more interesting, and basically amounts to implementing insertion sort on MutualOrd.

References

[ACD+18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, volume 10803 of *Lecture Notes in Computer Science*, pages 293–310, Heidelberg, Germany, 2018. Springer.

[BPT14] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Cardinals in Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 111–127, Heidelberg, Germany, 2014. Springer.

[Buc91] Wilfried Buchholz. Notation systems for infinitary derivations. *Archive for Mathematical Logic*, 30:227–296, 1991.

[Der93] Nachum Dershowitz. Trees, ordinals and termination. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 243–250, Heidelberg, Germany, 1993. Springer.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DR92] Nachum Dershowitz and Edward M. Reingold. Ordinal arithmetic with list structures. In Anil Nerode and Michael Taitslin, editors, *Logical Foundations of Computer Science*, volume 620 of *Lecture Notes in Computer Science*, pages 117–138, Heidelberg, Germany, 1992. Springer.

[Hes06] Gerhard Hessenberg. *Grundbegriffe der Mengenlehre*, volume 1. Vandenhoeck & Ruprecht, Göttingen, Germany, 1906.

[LS19] Peter Lefanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, pages 1–50, 2019.

[NF13] Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.

[NFXG20] Fredrik Nordvall Forsberg, Chuangjie Xu, and Neil Ghani. Three equivalent ordinal notation systems in Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20)*, pages 172–185. ACM, 2020.

[Sch77] Kurt Schütte. *Proof Theory*. Springer, Heidelberg, Germany, 1977.

[Sch17] Peter H. Schmitt. A mechanizable first-order theory of ordinals. In Renate Schmidt and Cláudia Nalon, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 10501 of *Lecture Notes in Computer Science*, pages 331–346, Heidelberg, Germany, 2017. Springer.

[Tak87] Gaisi Takeuti. *Proof Theory*. North-Holland Publishing Company, Amsterdam, 2 edition, 1987.

[Tur49] Alan Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, UK, 1949. University Mathematical Laboratory.

[UFP13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: a dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):87:1–87:29, 2019.

Shallow embedding of type theory is morally correct *

Ambrus Kaposi[1], András Kovács[1], and Nicolai Kraus[2]

[1] Eötvös Loránd University, Budapest, Hungary
akaposi|kovacsandras@inf.elte.hu
[2] University of Birmingham, United Kingdom
n.kraus@bham.ac.uk

Deep and shallow embeddings of monoids. In intensional type theory, when proving theorems that hold for every monoid, the usual method is assuming that there exists a pointed type with a binary operation and witnesses of some equalities (following Agda, we write $\equiv$ for the equality type and $=$ for definitional equality). We call this *deep embedding*, following the terminology for domain-specific languages [6].

$$\begin{array}{ll} \mathsf{M} & : \mathsf{Set} \\ \mathsf{u} & : \mathsf{M} \\ - \otimes - & : \mathsf{M} \to \mathsf{M} \to \mathsf{M} \end{array} \qquad \begin{array}{ll} \mathsf{ass} & : (a\,b\,c : \mathsf{M}) \to (a \otimes b) \otimes c \equiv a \otimes (b \otimes c) \\ \mathsf{idl} & : (a : \mathsf{M}) \to \mathsf{u} \otimes a \equiv a \\ \mathsf{idr} & : (a : \mathsf{M}) \to a \otimes \mathsf{u} \equiv a \end{array}$$

Combining the equalities idl, idr using congruence (ap) and transitivity, we prove the following example theorem.

$$\begin{aligned} &\mathsf{thm} : (a : \mathsf{M}) \to a \otimes (\mathsf{u} \otimes \mathsf{u}) \equiv a \\ &\mathsf{thm} := \lambda a.\mathsf{trans}\,(\mathsf{ap}\,(a \otimes -)\,(\mathsf{idl}\,\mathsf{u}))\,(\mathsf{idr}\,a) \end{aligned}$$

An alternative approach is *shallow embedding* of the monoid. Here we work with a *concrete* monoid such as the following one.

$$\begin{array}{ll} \mathsf{M} & := \mathsf{Bool} \to \mathsf{Bool} \\ \mathsf{u} & := \lambda x.x \\ a \otimes b & := \lambda x.a\,(b\,x) \end{array} \qquad \begin{array}{ll} \mathsf{ass}\,a\,b\,c & := \mathsf{refl}_{\lambda x.a\,(b\,(c\,x))} \\ \mathsf{idl}\,a & := \mathsf{refl}_a \\ \mathsf{idr}\,a & := \mathsf{refl}_a \end{array}$$

The advantage of using this monoid compared to a deeply embedded one is that the laws hold *definitionally*. For example, the proof of the above theorem now becomes trivial:

$$\begin{aligned} &\mathsf{thm} : (a : \mathsf{M}) \to a \otimes (\mathsf{u} \otimes \mathsf{u}) \equiv a \\ &\mathsf{thm} := \lambda a.\mathsf{refl}_a \end{aligned}$$

This monoid does not have more definitional equalities than a general monoid, e.g. we don't have the property that any two elements are equal (as would be the case if we used $\top \to \top$ as carrier). However, it has the property that there is an element propositionally unequal to u, e.g. $(\lambda a.\mathsf{true})$. Also, assuming function extensionality, we have $a \otimes a \otimes a \equiv a$ propositionally. These do not hold for every monoid, hence we can prove too many theorems.

*The first author was supported by by the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme, Project no. ED_18-1-2019-0030 (Application-specific highly reliable IT solutions), by the ÚNKP-19-4 New National Excellence Program of the Ministry for Innovation and Technology and by the Bolyai Fellowship of the Hungarian Academy of Sciences. The second author was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). The first and third authors were supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013 and EFOP-3.6.3-VEKOP-16-2017-00002). Finally, the third author acknowledges support by The Royal Society (URF\R1\191055).

We disallow such illegal constructions with an implementation hiding trick, using the record types of Agda with definitional η laws. The monoid is defined in the module Secret as a record M wrapping the function type Bool → Bool with constructor mkM and destructor unM. We only import Secret in the module Monoid and other modules are only allowed to import Monoid, but not Secret.

```
module Secret where
  record M := mkM {unM : Bool → Bool}
  u        := mkM (λx.x)
  a ⊗ b    := mkM (λx.unM a (unM b x))
```

```
module Monoid where
  import Secret privately
  M      := Secret.M
  u      := Secret.u
  _⊗_    := Secret._⊗_
```

A module importing Monoid only has access to M, u and $-\otimes-$, but not mkM and unM. However, the definitional behaviour of the operations is exported, so proofs are still as easy as for the naive shallow embedding.

In principle, we should be able to transfer any proof about the shallowly embedded monoid to a deeply embedded one.

Deep and shallow embeddings of type theory. Type theory can also be seen as an algebraic structure [5]. Compared to monoids, there are more sorts and many more operations and equations. Metatheoretic proofs about type theory can be seen as constructions on models of type theory. This is the case e.g. for normalisation [1], parametricity [2], or canonicity [4], the latter two being special cases of gluing [7], a construction on a weak homomorphism of models. When formalising such arguments for deeply embedded models, combining equalities and transporting over them becomes a huge bureaucratic burden, sometimes called "transport hell". However, as in the case of monoids, we are able to reuse properties of our metatheory (e.g. strict associativity of function composition) to define a shallow embedding of type theory, where (most) equalities are definitional. The shallow embedding is a concrete model (the *standard model* [2] – sometimes called *set model* or *metacircular interpretation*) in which all equations hold definitionally, and as before, we only export the interface. In this model, contexts are defined as Set, a type over Γ is a $\Gamma \to$ Set function, terms have dependent function type $(\gamma : \Gamma) \to A\,\gamma$.

Moral correctness. By proving all equations using refl, we can check that our shallow embedding has enough equalities. The implementation hiding makes sure that we cannot construct too many elements and proofs. However, we have to prove that we don't have too many definitional equalities. When showing this, we assume that Agda implements type theory correctly and we look at the standard model from outside of Agda. Externally, contexts are given by Tm · U (Agda-terms in the empty context of type U for the universe), types are in Tm · (El Γ ⇒ U), terms are in Tm · (Π (El Γ) (El (A $ var 0))), and so on. We prove that for any two syntactic terms, if their external standard interpretations are definitionally equal, then they are also definitionally equal. This shows that the standard model does not add more equalities than there are in the syntax.

Applications. Using this form of shallow embedding, for a type theory with an infinite hierarchy of universes, Π, Σ, Bool and Id types, we formalised [8] the syntactic logical predicate interpretation of type theory [3], canonicity [4], and gluing [7]. The line count of the parametricity proof is roughly 20% of the same proof for the deeply embedded syntax [2]. More details can be found in a paper presented at MPC 2019 [9].

References

[1] Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for dependent types. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodik and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016.

[3] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012.

[4] Thierry Coquand. Canonicity and normalization for dependent type theory. *Theor. Comput. Sci.*, 777:184–191, 2019.

[5] Peter Dybjer. Internal type theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.

[6] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). *SIGPLAN Not.*, 49(9):339–347, August 2014.

[7] Ambrus Kaposi, Simon Huber, and Christian Sattler. Gluing for type theory. In Herman Geuvers, editor, *Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, 2019.

[8] Ambrus Kaposi, András Kovács, and Nicolai Kraus. Formalisations in Agda using a morally correct shallow embedding, May 2019.

[9] Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory is morally correct. In Graham Hutton, editor, *Mathematics of Program Construction*, pages 329–365, Cham, 2019. Springer International Publishing.

A type system for simple quantum processes

Vítor Fernandes[1], Renato Neves[2], and Luís Barbosa[2]

[1] University of Minho, Braga, Portugal
vegf17@gmail.com
[2] International Iberian Nanotechnology Laboratory, INESC-TEC & University of Minho, Braga, Portugal
{nevrenato,lsb}@di.uminho.pt

Motivation and context. The development of quantum mechanics in the last century offers the possibility of building computers ruled by quantum laws that perform much better in certain tasks than the classical counterparts. Examples of such tasks include unstructured searching and factorization of integers [6]. Not only this, quantum mechanics also gave rise to new communication protocols, such as the quantum teleportation and the BB84 communication protocols [6].

The few existing quantum computers adopt a QRAM architecture [4]: in a nutshell, a master-slave architecture in which a classical computer (the master) handles a complex task by, among other things, selecting some highly costly computational subtasks and requesting a quantum computer (the slave) to solve them. Such an architecture is necessary, because quantum computers become increasingly unreliable as their computational tasks grow in length and (quantum) memory requirements. The interaction between classical and quantum computers in this architecture highlights the importance of considering concurrency and communication in models where both classical and quantum features are present. In particular, it seems relevant to extend the theory of process algebra to the quantum domain.

Quantum process algebras have already been introduced in the past, two main examples being qCCS [1] and CQP [3]. Both accommodate classical and quantum operations, and thus can be used to study the architecture above. The presence of quantum features demands certain syntactic restrictions for ensuring *physical realisability of processes* and which can take complex form. The prime example is measurement of qubits (the basic information unit in quantum computing): measuring a qubit essentially destroys it. Another interesting case, arising from communication, is qubit communication: once a qubit (a physical resource) is sent away by a process the latter no longer has access to it.

Goal. Our goal is to study the role of type systems in ensuring physical realisability of processes. Here we discuss measurement and qubit communication, but we are also particularly interested on quantum decoherence: a qubit stores information only for very short periods of time.

Contributions. CQP is a process algebra with quantum features in the spirit of π-calculus. qCCS, on the other hand, is a simpler quantum process algebra in the spirit of (value-passing) CCS. The latter is our object of study. We present a type system (more formally, a derivation system) for qCCS to ensure physical realisability (*i.e.* implementable in quantum systems) of processes in what concerns measurement and qubit communication; subsequently, we prove type preservation and weakening. We regard these results as a basic stepping stone for achieving the goal described above. Details of this work are found in [2, 7].

CQP already has a type system, but it does not support recursion.

Quantum CCS. qCCS is an extension of value-passing CCS [5] with operations for manipulating qubits (specifically, super-operators and measurement). A fragment of its syntax is given by the grammar,

$$\mathtt{P},\mathtt{Q} \ ::= \varepsilon[\tilde{q}].\mathtt{P} \mid c?x.\mathtt{P} \mid c!e.\mathtt{P} \mid \mathtt{c}?q.\mathtt{P} \mid \mathtt{c}!q.\mathtt{P} \mid \mathtt{M}[q;x].\mathtt{P} \mid \mathtt{P}+\mathtt{Q} \mid \mathtt{P} \,||\, \mathtt{Q}$$

The expression $\varepsilon[\tilde{q}]$ represents the action of a super-operator ε on a finite list of qubits $\tilde{q}$. The letter c represents a classical communication channel, $\mathtt{c}$ a quantum communication channel, x a classical variable, e an arithmetic expression over Real, q a quantum variable, and $\mathtt{M}[q;x]$ represents the measurement of qubit q with the result stored on a classical variable x. The other constructs are standard in process algebra. As it is, the grammar allows processes such as $\mathtt{c}!q.\varepsilon[q]$ – which reads "send the qubit q via channel $\mathtt{c}$ and then apply operator ε to q" – and processes such as $\mathtt{M}[q;x].\mathtt{c}!q$ which reads "measure qubit q and then send it away". As discussed before, these processes are physically unrealisable. The authors of QCCS fix this by restricting composition of processes on basis of their free quantum variables [1]. Our contribution is in essence the extraction of a type system from these restrictions.

Type system. The type system relative to the fragment of the grammar above is as follows:

$$\frac{\Gamma \vdash \mathtt{P} \quad \tilde{q} \subseteq \Gamma^Q}{\Gamma \vdash \varepsilon[\tilde{q}].\mathtt{P}}\ \textbf{(OP)} \qquad \frac{\Gamma, x : C \vdash \mathtt{P}}{\Gamma \vdash c?x.\mathtt{P}}\ \textbf{(C-IN)} \qquad \frac{\Gamma \vdash \mathtt{P} \quad fv(e) \subseteq \Gamma^C}{\Gamma \vdash c!e.\mathtt{P}}\ \textbf{(C-OUT)}$$

$$\frac{\Gamma, x : C \vdash \mathtt{P}}{\Gamma, q : Q \vdash \mathtt{M}[q;x].\mathtt{P}}\ \textbf{(MEAS)} \qquad \frac{\Gamma, q : Q \vdash \mathtt{P}}{\Gamma \vdash \mathtt{c}?q.\mathtt{P}}\ \textbf{(Q-IN)} \qquad \frac{\Gamma \vdash \mathtt{P}}{\Gamma, q : Q \vdash \mathtt{c}!q.\mathtt{P}}\ \textbf{(Q-OUT)}$$

$$\frac{\Gamma_1 \vdash \mathtt{P} \quad \Gamma_2 \vdash \mathtt{Q}}{\Gamma_1 \cup \Gamma_2 \vdash \mathtt{P} + \mathtt{Q}}\ \textbf{(SUM)} \qquad \frac{\Gamma_1 \vdash \mathtt{P} \quad \Gamma_2 \vdash \mathtt{Q} \quad \Gamma_1^Q \cap \Gamma_2^Q = \emptyset}{\Gamma_1 \cup \Gamma_2 \vdash \mathtt{P} \mid\mid \mathtt{Q}}\ \textbf{(COMM)}$$

where Γ is a non-repetitive list of quantum variables $q : Q$ and classical variables $x : C$. Given a typing context Γ we denote its projection on quantum variables by Γ^Q and on classical variables by Γ^C. The judgement $\Gamma \vdash P$ reads "from the variables in Γ we build the process P". In detail, rule **(OP)** corresponds to the application of a super-operator to a list of qubits $\tilde{q}$ which must exist in Γ. Rule **(Q-OUT)** corresponds to sending qubit q away via channel $\mathtt{c}$, and forbids $\mathtt{P}$ from accessing it by ensuring that q is not in Γ. Rule **(MEAS)** represents measuring a qubit q; similarly to the previous rule it ensures that $\mathtt{P}$ has no access to the qubit. Rule **(COMM)** ensures that processes $\mathtt{P}$,$\mathtt{Q}$ cannot run in parallel with shared quantum variables. Similarly to communicating qubits and measurement, this restriction arises from the fact that qubits need to be treated as physical resources. For example, without the restriction we would be able to write $\mathtt{M}[q;x].\, c!ok \cdots \mid\mid c?ok.\, \varepsilon(q) \ldots$, a process not physically realisable because the process on the left destroys q but the process on the right acts as if the qubit still exists. The others rules in the type system are standard.

The restriction on shared variables echoes the famous topic of mutual exclusion in accessing critical memory sections [5]. A very distinctive feature in the quantum case, however, is that even if qubits are not shared, one process can still change the qubits of the other by resorting to entanglement [6].

Some properties of the type system. The following properties of the type system proposed are proved by induction. See the details in [2, 7].

Theorem 1 (Weakening)**.** Assume that $\Gamma \vdash \mathtt{P}$ and let x be either a classical or a quantum variable in Γ. Then if x is a classical variable (resp. a quantum variable) it is true that $\Gamma, x : C \vdash \mathtt{P}$ (resp. $\Gamma, x : Q \vdash \mathtt{P}$)

Theorem 2 (Type preservation)**.** Assume that $\Gamma \vdash \mathtt{P}$ (in words, that $\mathtt{P}$ is physically realisable). Then every process reached by executing $\mathtt{P}$ is also physically realisable.

The last theorem and rule **(COMM)** ensure that qubits are never shared simultaneously along the parallel execution of two processes.

References

[1] Yuan Feng, Runyao Duan, and Mingsheng Ying. Bisimulation for quantum processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(4):17, 2012.

[2] Vitor Fernandes. Integration of time in a quantum process algebra. Master's thesis, Dep. Informatica, Universidade do Minho, 2019. Available at https://github.com/vegf17/dissertation.

[3] Simon J Gay and Rajagopal Nagarajan. Types and typechecking for communicating quantum processes. *Mathematical Structures in Computer Science*, 16(3):375–406, 2006.

[4] Emmanuel Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Lab., NM (United States), 1996.

[5] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[6] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge, 2000.

[7] Luis Barbosa Vitor Fernandes, Renato Neves. Proof of type preservation. Technical report, 2020. Available at https://github.com/vegf17/dissertation.

6 Types, proofs and programs (WG3)

Dependent Type Theory in Polarised Sequent Calculus

Étienne Miquey[1], Xavier Montillet[2], and Guillaume Munch-Maccagnoni[2]

[1] CNRS, LSV, ÉNS Paris-Saclay, Inria
[2] Inria, LS2N CNRS

Thanks to several works on classical logic in proof theory, it is now well-established that continuation-passing style (CPS) translations in call by name and call by value correspond to different polarisations of formulae (Girard, 1991; Danos, Joinet, and Schellinx, 1997; Laurent, 2002). Extending this observation, the last author proposed a term assignment for a polarised sequent calculus (where the polarities of formulae determine the evaluation order), in which various calculi from the literature can be obtained with *macros* responsible for the choices of polarities (Munch-Maccagnoni, 2013, III). It explains various CPS translations from the literature by means of a decomposition through a single CPS for sequent calculus. This calculus has later proved to be a fruitful setting to study the addition of effects and resource modalities (Curien, Fiore, and Munch-Maccagnoni, 2016), providing a categorical proof theory of *Call By Push Value* semantics (Levy, 2004).

We propose to bring together a dependently-typed theory (ECC) and polarised sequent calculus, by presenting a calculus $\mathbf{L_{dep}}$ suitable as a vehicle for compilation and representation of effectful computations. As a first step in that direction, we show that $\mathbf{L_{dep}}$ advantageously factorize a dependently typed continuation-passing style translation for **ECC**+`call/cc`. To avoid the inconsistency of type theory with control operators, we restrict their interaction. Nonetheless, in the pure case, we obtain an unrestricted translation from **ECC** to itself, thus opening the door to the definition of dependently typed compilation transformations.

Overview of $\mathbf{L_{dep}}$ Recall that the key notion of term assignments for sequent calculi is that of a *command*, written $\langle t \parallel e \rangle$, which can be understood as a state of an abstract machine, representing the evaluation of an proof (or *expression*) t against a counter-proof e that we call *context*. Their typing judgements are of the form $\Gamma \vdash t : A \mid \Delta$ and $\Gamma \mid e : A \vdash \Delta$, which correspond respectively to underlying sequents $\Gamma \vdash A, \Delta$ and $\Gamma, A \vdash \Delta$, in which A is in both cases the *principal formula* of the sequent. The command $\langle t \parallel e \rangle$ is the result of applying the cut rule with t and e as premises: $\langle t \parallel e \rangle : (\Gamma \vdash \Delta)$. It represents a cut rule with no principal formula.

But, in comparison to other presentations of sequent calculi, and like in Girard's original formulation of **LC**, our logic features a negation operator $\cdot^{\perp}$ which is involutive strictly: $A = A^{\perp\perp}$. This involution allows us to represent any sequent $c : (\Gamma \vdash \Delta)$ (resp. $\Gamma \vdash t : A \mid \Delta$) as a sequent $c : (\vdash \Gamma^{\perp}, \Delta)$ (resp. $\vdash \Gamma^{\perp}, \Delta \mid t : A$) with all formulae on the right. Thus, we are able to use a single grammar to describe both expressions and contexts.

The sequent calculus we propose is, in term of expressiveness, an extension of Luo's ECC. Namely, ECC contains dependent products $\Pi(x : A).B$ (becoming here a dependent $\parr$) and dependent sums $\Sigma(x : A).B$ (becoming here a dependent $\otimes$), a cumulative hierarchy of universes $\square_i$ and an impredicative propositional universe $\mathbb{P}$, the inductive type of booleans with dependent elimination $\mathbb{B}$, and equalities between terms $t = u$:

Atoms	$C ::= x \mid \mathbb{B} \mid \mathbb{P} \mid \square_i \mid t = u$	**Values**	$V ::= x \mid A \mid V \otimes_A V' \mid \texttt{true} \mid \texttt{false} \mid \texttt{refl}$
Types^{+}	$P ::= C \mid A \otimes x.B \mid \Downarrow A$		$\mid \mu(x \otimes_A y).c \mid \mu^{\ominus}x.c \mid \mu[c_1 \mid c_2] \mid \mu{=}c \mid \hat{\mu}c$
Types$^{\ominus}$	$N ::= C^{\perp} \mid \parr(A).xB \mid \Uparrow A$	**Terms**	$t ::= \mu^{+}x.c \mid \hat{} \mid V^{\diamond}$
Types	$A ::= P \mid N$	**Commands**	$c ::= \langle t \parallel V \rangle^{+}$

where the notations $\mu^{+}x.c / \mu^{\ominus}x.c$ distinguish the binder according to the polarity of the corresponding type.

Since sequent calculi allow us to manipulate classical logic, we need to restrict dependencies to avoid logical inconsistencies (Herbelin, 2005). Following previous works (Herbelin, 2012; Miquey, 2019), we only allow *negative-elimination-free* (NEF) terms within types, which are *thunkable* (value-like) terms. In fact, we relax this constraint into that of Girard's stoup (Girard, 1991), which similarly implies thunkability/linearity (Munch-Maccagnoni, 2013, IV.6). We take advantage of delimited control operators (in the form of $\hat{\mu}c$ and $\star$) to separate regular and dependent typing modes:

$$\frac{\vdash \Gamma \mid t : P \qquad \vdash \Gamma \mid V : P^\perp}{\langle t \parallel V \rangle^+ : (\vdash \Gamma)} \qquad \frac{\vdash \Gamma \mid t : P \qquad \vdash_{B[\bullet]} \Gamma \mid V : P^\perp}{\langle t \parallel V \rangle^+ : (\vdash_{B[t]} \Gamma)}\ t \in \mathsf{NEF}$$

$$\frac{c : (\vdash \Gamma, x : N)}{\vdash \Gamma \mid \mu^\ominus x.c : N} \qquad \frac{c : (\vdash_N \Gamma)}{\vdash \Gamma \mid \hat{\mu} c : N} \qquad \frac{c : (\vdash_{B[x]} \Gamma, x : N)}{\vdash_{B[\bullet]} \Gamma \mid \mu^\ominus x.c : N} \qquad \frac{\bullet \notin B}{\vdash_B \Gamma \mid \star : B^\perp}$$

Regular mode **Dependent mode**

Observe that in the latter, the turnstile is annotated with a return type whose dependencies evolve with the typing derivation (see Miquey 2019 for more details). For instance, considering the type:

$$T(b) = \mu^+ x.\left\langle b \,\middle\|\, \mu[\langle \mathbb{P} \parallel x\rangle^+ \mid \langle \mathbb{B} \parallel x\rangle^+]\right\rangle^+$$

which verifies that $T(\mathtt{true}) \equiv \mathbb{P}$ and $T(\mathtt{false}) \equiv \mathbb{B}$, we can inhabit it with the following term:

$$\dfrac{\vdash b : \mathbb{B}^\perp \mid b : \mathbb{B} \qquad \dfrac{\dfrac{\vdash \mid \Pi(X : \mathbb{P}).X : \Uparrow T(\mathtt{true})}{\langle \Pi(X : \mathbb{P}).X \parallel \star\rangle^\ominus : (\vdash_{\Uparrow T(\mathtt{true})})} \qquad \dfrac{\vdash \mid \mathtt{true} : \Uparrow T(\mathtt{false})}{\langle \mathtt{true} \parallel \star\rangle^\ominus : (\vdash_{\Uparrow T(\mathtt{false})})}}{\vdash_{\Uparrow T(\bullet)} b : \mathbb{B}^\perp \mid \mu[\langle \Pi(X : \mathbb{P}).X \parallel \star\rangle^\ominus \mid \langle \mathtt{true} \parallel \star\rangle^\ominus] : \mathbb{B}^\perp}}{\dfrac{\left\langle b \,\middle\|\, \mu[\langle \Pi(X : \mathbb{P}).X \parallel \star\rangle^\ominus \mid \langle \mathtt{true} \parallel \star\rangle^\ominus]\right\rangle^+ : (\vdash_{\Uparrow T(b)} b : \mathbb{B}^\perp)}{\vdash b : \mathbb{B}^\perp \mid \hat{\mu}\left\langle b \,\middle\|\, \mu[\langle \Pi(X : \mathbb{P}).X \parallel \star\rangle^\ominus \mid \langle \mathtt{true} \parallel \star\rangle^\ominus]\right\rangle^+ : \Uparrow T(b)}}$$

CPS translations for ECC Following the approach advocated in Boulier, Pédrot, and Tabareau (2017), the soundness of our system is proved by means of a syntactic model. In other words, we define a typed translation from our system to (an extension of) Luo's **ECC** Luo (1990). In broad lines, this translation follows the structure of the call-by-value continuation-passing style translation highlighted in Miquey (2019): we use dependent and parametric return types for continuations, and we translate NEF terms t at two different levels $[t]_0$ and $[t]_1$ in a way that is reminiscent of parametricity translations. For instance, the translations of a (closed and NEF) b boolean verify:

$$[b]_1 \ : \Pi(R : \mathbb{B} \to \mathbb{P}).((\Pi(x : \mathbb{B}).R\,x) \to R\,[b]_0))$$

Observe that by parametricity, this implies in particular that for any continuation k of parametric return type R, we have $[b]_1\, R\, k \equiv k\,[b]_0$, emphasizing that such a translation is only compatible with NEF terms that observationally behave like values.

Insofar as we can easily embed **ECC**+`call/cc` (evaluated in call by value) in our system, this translation allows us to factorize a CPS translation from this calculus to the (pure) **ECC**:

$$\mathbf{ECC} + \mathtt{call/cc} \xrightarrow{\text{macros}} \mathbf{L_{dep}} \xrightarrow{\text{CPS}} \mathbf{ECC}$$

Interestingly, by considering only the pure (by-value) **ECC**, we can define a dependently typed translation to itself without any kind of restriction on dependent types[1]. Our translation improves over Bowman, Cong, Rioux, and Ahmed (2017) in that no extra assumption (in particular, we do not require an extensional type theory) are necessary to prove its soundness.

[1] A Coq development formalizing some aspects of these ideas is available at: https://www.irif.fr/~emiquey/content/CPS_ECC.v

References

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *CPP*. https://doi.org/10.1145/3018610.3018620

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-Preserving CPS Translation of Î£ and Î Types is Not Not Possible. *Proc. ACM Program. Lang.* 2, POPL, Article Article 22, 33 pages. https://doi.org/10.1145/3158110

Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. 2016. A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. In *Proceedings of POPL '16*. https://doi.org/10.1145/2837614.2837652

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. 1997. A new deconstructive logic: linear logic. *Journal of Symbolic Logic* (1997). https://doi.org/10.2307/2275572

Jean-Yves Girard. 1991. A new constructive logic: classic logic. *Mathematical Structures in Computer Science* (1991). https://doi.org/10.1017/S0960129500001328

Hugo Herbelin. 2005. On the Degeneracy of Sigma-Types in Presence of Computational Classical Logic. In *Proceedings of TLCA 2005 (LNCS)*, Pawel Urzyczyn (Ed.), Vol. 3461. Springer, 209–220. https://doi.org/10.1007/11417170_16

Hugo Herbelin. 2012. A Constructive Proof of Dependent Choice, Compatible with Classical Logic. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 365–374. https://doi.org/10.1109/LICS.2012.47

Olivier Laurent. 2002. *Étude de la polarisation en logique*. Thèse de Doctorat. Université Aix-Marseille II. https://tel.archives-ouvertes.fr/tel-00007884

Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Kluwer Academic Publishers. https://doi.org/10.1007/978-94-007-0954-6

Zhaohui Luo. 1990. *An Extended Calculus of Constructions*. PhD Thesis. University of Edinburgh. https://era.ed.ac.uk/bitstream/handle/1842/12487/Luo1990.Pdf

Étienne Miquey. 2019. A Classical Sequent Calculus with Dependent Types. *ACM Transactions on Programming Languages and Systems* 41 (2019). https://doi.org/10.1145/3230625

Guillaume Munch-Maccagnoni. 2013. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. Theses. Université Paris-Diderot - Paris VII. https://tel.archives-ouvertes.fr/tel-00918642

On the logical structure of choice and bar induction principles

Nuria Brede and Hugo Herbelin

[1] University of Potsdam, Germany
nuria.brede@uni-potsdam.de
[2] IRIF, CNRS, Université de Paris, Inria, France
hugo.herbelin@inria.fr

We develop an approach to choice principles and to their bar-induction contrapositive principles, as extensionality schemes connecting an "operational" or "intensional" view of respectively ill-foundedness and well-foundedness properties to an "idealistic" or "observational" view of these properties. In a first part, we classify and analyse the relations between different intensional definitions of countable ill-foundedness and countable well-foundedness involving Bar Induction, Dependent Choice, König's Lemma and the Fan Theorem. In a second part, we integrate the Ultrafilter Lemma to the picture and develop, for A a domain, B a codomain and T a "filter" on finite approximations of functions from A to B, a "filtered" form of axiom of choice ACF_{ABT} and dually of a generalised bar induction principle GBI_{ABT} such that, writing $\mathbb{N}$ and $\mathbb{B}\mathsf{ool}$ for the types of natural numbers and Booleans values respectively, we have:

- ACF_{ABR^*} intuitionistically captures the strength of the general axiom of choice expressed as $\forall a^A\, \exists b^B\, R(a,b) \to \exists \alpha\, \forall a\, R(a,\alpha(a)))$, where R^* is an "unconstraining" filter deriving pointwise from the relation R
- $ACF_{A\mathbb{B}\mathsf{ool}T}$ intuitionistically captures the Boolean Maximal Ideal Theorem / Ultrafilter Lemma
- $ACF_{\mathbb{N}BT}$ intuitionistically captures the axiom of dependent choice
- $ACF_{\mathbb{N}\mathbb{B}\mathsf{ool}T}$ captures the (choice) strength of Weak König's Lemma (up to weak classical reasoning)
- $GBI_{A\mathbb{B}\mathsf{ool}T}$ intuitionistically captures Gödel's completeness theorem in the form validity implies provability
- $GBI_{\mathbb{N}BT}$ intuitionistically captures the strength of Bar Induction principles
- $GBI_{\mathbb{N}\mathbb{B}\mathsf{ool}T}$ intuitionistically captures the (choice) strength of the Weak Fan Theorem

For classifying the countable case, we use the definitions below, which all apply to a predicate T over the set A^* of finite sequences of elements of a given domain A. Our focus being purely logical, we do not impose any arithmetical restriction (such as decidability) on the predicate.

We use the letter a to range over elements of A, the letter u to range over the elements of A^*, n to range over the natural numbers $\mathbb{N}$ and α to range over functions from $\mathbb{N}$ to A. The empty sequence is denoted $\langle\rangle$ and sequence extension $u \star a$.

Equivalent concepts on dual predicates	
T is a tree $\forall u\, \forall a\, (u \star a \in T \to u \in T)$	T is monotone $\forall u\, \forall a\, (u \in T \to u \star a \in T)$
T is progressing $\forall u\, (u \in T \to \exists a\, u \star a \in T)$	T is hereditary $\forall u\, ((\forall a\, u \star a \in T) \to u \in T)$

Dual concepts on dual predicates	
ill-foundedness-style	*well-foundedness-style*
Intensional concepts	
T has unbounded paths $\forall n\,\exists u\,(\lvert u\rvert = n \wedge \forall v\,(v \leq u \to v \in T))$	T is uniformly barred $\exists n\,\forall u\,(\lvert u\rvert = n \to \exists v\,(v \leq u \wedge v \in T))$
T is staged infinite[1] $\forall n\,\exists u\,(\lvert u\rvert = n \wedge u \in T)$	T is staged barred[1] $\exists n\,\forall u\,(\lvert u\rvert = n \to u \in T)$
T is a spread $\langle\rangle \in T \wedge T$ progressing	T is resisting[1] T hereditary $\to \langle\rangle \in T$
pruning of T $\nu X.\lambda u.(u \in T \wedge \exists a\, u \star a \in X)$	adherence[1] of T $\mu X.\lambda u.(u \in T \vee \forall a\, u \star a \in X)$
T is productive[1] $\langle\rangle \in$ pruning of T	T is inductively barred $\langle\rangle \in$ adherence of T
Observational concepts	
T has an infinite branch $\exists \alpha\,\forall u\,(u$ initial segment of $\alpha \to u \in T)$	T is barred $\forall \alpha\,\exists u\,(u$ initial segment of $\alpha \wedge u \in T)$

Different equivalences will be stated about the intensional definitions, under either a classical (i.e. interpreting $\exists$ classically), intuitionistic (i.e. interpreting $\exists$ intuitionistically), or linear reading (i.e. additionally interpreting $\to$ as a linear arrow and conjunction as a tensor product). (Note [1]: not being aware of an established terminology, we use here our own terminology.)

Then, we shall take the definition of inductively barred, productive, having an infinite branch and barred as references and, for T over A^*, define:

Tree-based choice principles and bar induction principles	
(A variant of) Dependent Choice (DC^{prod})	Bar Induction (BI^{ind})
T is productive $\to$ T has an infinite branch	T barred $\to$ T inductively barred

The naive generalisation to the non-countable case leads to the following definitions which satisfy the logical equivalences of the introduction, where $\downarrow T$ and $\uparrow T$ denote respectively the tree and monotone closures of a predicate T over $(A \times B)^*$, and $\prec$ is finitely approximating:

Dual concepts on dual predicates	
ill-foundedness-style	*well-foundedness-style*
Intensional concepts	
T coinductively A-B-choosable from u $\nu X.\lambda u.\left(\begin{array}{l} u \in \downarrow T \\ \wedge\, \forall\, a \notin dom(u)\, \exists b\, u \star (a,b) \in X \end{array}\right)$	T inductively A-B-barred from u $\mu X.\lambda u.\left(\begin{array}{l} u \in \uparrow T \\ \vee\, \exists\, a \notin dom(u)\, \forall b\, u \star (a,b) \in X \end{array}\right)$
Extensional concepts	
T is A-B-choosable $\exists \alpha \in (A \to B)\, \forall u\,(u \prec \alpha \to u \in T)$	T is A-B-barred $\forall \alpha \in (A \to B)\, \exists u\,(u \prec \alpha \wedge u \in T)$

Dual axioms	
Naive Filtered Axiom of Choice (ACF^{naive}_{ABT})	Naive Generalized Bar Induction (GBI^{naive}_{ABT})
T coinductively A-B-choosable from $\langle\rangle$ implies T A-B-choosable	T A-B-barred implies T inductively A-B barred from $\langle\rangle$

Unfortunately, the naive form of ACF_{ABT} and GBI_{ABT} is conjectured inconsistent. We shall then propose an alternative to make it exactly the strength of the general axiom of choice while preserving the logical equivalences obtained by restricting A, B or T.

On the Proof Theory of Property-Based Testing of Coinductive Specifications, or: PBT to Infinity and beyond

Roberto Blanco[1], Dale Miller[2], and Alberto Momigliano[3]

[1] INRIA Paris, France
[2] INRIA Saclay & LIX, École Polytechnique, France
[3] DI, Università degli Studi di Milano, Italy

Reasoning about infinite computations via coinduction and corecursion has an ever increasing relevance in formal methods and, in particular, in the semantics of programming languages, starting from [16]; see also [13] for a compelling example — and, of course, coinduction underlies (the meta-theory of) process calculi. This was acknowledged by researchers in proof assistants, who promptly provided support for coinduction and corecursion from the early 90's on, see [19, 10] for the beginning of the story concerning the most popular frameworks.

It also became apparent that tools that search for refutations/counter-examples of conjectures prior to attempting a formal proof are invaluable: this is particularly true in PL theory, where proofs tend to be shallow but may have hundreds of cases. One such approach is *property-based testing* (PBT), which employs automatic test data generation to try and refute executable specifications. Pioneered by *QuickCheck* for functional programming [7], it has now spread to most major proof assistants [4, 18].

In general, PBT does not extend well to coinductive specifications (an exception being Isabelle's *Nitpick*, which is, however, a counter-model generator). A particular challenge, for example, for *QuickChick* is extending it to work with Coq's notion of coinductive via *guarded* recursion (which is generally seen to be an unsatisfactory approach to coinduction). We are not aware of applications of PBT to other form of coinduction, such as *co-patterns* [1].

While PBT originated in the functional programming community, we have given in a previous paper ([5]) a reconstruction of some of its features (operational semantics, different flavors of generation, shrinking) in purely proof-theoretic terms employing the framework of *Foundational Proof Certificates* [6]: the latter, in its full generality, defines a range of proof structures used in various theorem provers such as resolution refutations, Herbrand disjuncts, tableaux, etc. In the context of PBT, the proof theory setup is much simpler. Consider an attempt to find counter-examples to a conjecture of the form $\forall x[(\tau(x) \wedge P(x)) \supset Q(x)]$ where τ is a typing predicate and P and Q are two other predicates defined using Horn clause specifications. By negating this conjecture, we attempt to find a (focused) proof of $\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)]$. In the focused proof setting, the *positive phase* (where test cases are generated) is represented by $\exists x$ and $(\tau(x) \wedge P(x))$. That phase is followed by the *negative phase* (where conjectured counter-examples are tested) and is represented by $\neg Q(x)$. FPCs are simple logic programs that guide the search for potential counter-examples using different generation strategies; they further capture diverse features such as δ-debugging, fault isolation, explanation, etc. Such a range of features can be programmed as the *clerks and experts* predicates that decorate the sequent rules used in a FPC proof checking kernel: the kernel is also able to do a limited amount of proof reconstruction.

As explained in [5], the standard PBT setup needs little more than Horn logic. However, when addressing infinite computations, we need richer specifications. While coinductive logic programming, see [21] and [3] for a much more principled and in depth treatment, may at first

seem to fit the bill, the need to model infinite *behavior* rather than infinite objects, that is (ir)rational terms on the domain of discourse, has lead us to adopt a much stronger logic (and associated proof theory) with explicit rules for induction and coinduction.

A natural choice for such a logic is the fixed point logic $\mathcal{G}$ [9] and its linear logic cousin μMALL [2], which are associated to the *Abella* proof assistant and the *Bedwyr* model-checker. In fact, the latter has already been used for related aims [11].

To make things more concrete, consider the usual rules for CBV evaluation in the λ-calculus with constants, but define it *coinductively*, following see [13]: using Bedwyr's concrete syntax, this is written as:

```
Define coinductive coeval: tm -> tm -> prop by
 coeval (con C) (con C);
 coeval (fun R) (fun R);
 coeval (app M N) V :=
   exists R W, coeval M (fun R) /\ coeval N W /\ coeval (R W) V.
```

Is evaluation still deterministic? And if not, can we find terms `E`, `V1`, and `V2` such that `coeval E V1 /\ coeval E V2 /\ (V1 = V2 -> false)`?[1] Indeed we can, since a divergent term such as Ω co-evaluates to anything. In fact, co-evaluation is not even type sound in its generality. Our PBT approach aims to find such counter-examples.

It can also be used to separate various notion of equivalences in lambda and process calculi: for example, separating applicative and ground similarity in PCFL [20], or analogous standard results in the π-calculus. While analogous goals have been achieved for labeled transition systems and for CCS (using, for example, the *Concurrency Workbench*), it is a remarkable feature of the proof-theoretic account that is easy to generalizes PBT from a system without bindings (say, CCS) to a system with bindings (say, the π-calculus). Such ease is possible since proof theory accommodates the *λ-tree syntax* approach to treating bindings [14]: this approach includes the ∇ quantifier [15] that appears in both Abella and Bedwyr.

In our current setup, we attempt to find counter-examples, using Bedwyr to execute both the generation of test cases (controlled by using specific FPCs [5]) and the testing phase. Such an implementation of PBT has the advantages of allowing us to piggyback on Bedwyr's facilities for efficient proof search via tabling for (co)inductive predicates. There are a couple of treatments of the negation in the testing phase. One approach to eliminating negation from intuitionistic specification can be based on the techniques in [17]. Another approach identifies the proof theory behind model checking as the linear logic μMALL [12] and in that setting, negations can be eliminated by using De Morgan duality (and inequality).

References

[1] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *POPL*, pages 27–38. ACM, 2013.

[2] D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. Comput. Log.*, 13(1):2:1–2:44, 2012.

[3] H. Basold, E. Komendantskaya, and Y. Li. Coinduction in uniform: Foundations for corecursive proof search with horn clauses. In *ESOP*, volume 11423 of *Lecture Notes in Computer Science*, pages 783–813. Springer, 2019.

[1] In this case equality is purely syntactical, since by construction terms will be ground when compared, but the logic implements a richer notion [8].

[4] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS*, volume 6989 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2011.

[5] R. Blanco, D. Miller, and A. Momigliano. Property-based testing via proof reconstruction. In *PPDP*, pages 5:1–5:13. ACM, 2019.

[6] Z. Chihani, D. Miller, and F. Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 59(3):287–330, 2017.

[7] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.

[8] A. Gacek, D. Miller, and G. Nadathur. Nominal abstraction. *Inf. Comput.*, 209(1):48–73, 2011.

[9] A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2):241–273, Aug 2012.

[10] E. Giménez. Codifying guarded definitions with recursion schemes. In P. Dybjer and B. Nordström, editors, *Selected Papers 2nd Int. Workshop on Types for Proofs and Programs, TYPES'94, Båstad, Sweden, 6–10 June 1994*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer-Verlag, Berlin, 1994.

[11] Q. Heath and D. Miller. A framework for proof certificates in finite state exploration. In *PxTP@CADE*, volume 186 of *EPTCS*, pages 11–26, 2015.

[12] Q. Heath and D. Miller. A proof theory for model checking. *J. Autom. Reasoning*, 63(4):857–885, 2019.

[13] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[14] D. Miller. Mechanized metatheory revisited. *Journal of Automated Reasoning*, Oct. 2018.

[15] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.

[16] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[17] A. Momigliano. Elimination of negation in a logical framework. In *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 411–426. Springer, 2000.

[18] Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.

[19] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, Apr. 1997.

[20] A. M. Pitts. Operationally Based Theories of Program Equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, 1997.

[21] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In L. Arge, C. Cachin, T. Jurdziński, and A. Tarlecki, editors, *Automata, Languages and Programming*, pages 472–483, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

Duality in intuitionistic propositional logic

Paweł Urzyczyn

Institute of Informatics, University of Warsaw, Poland
urzy@mimuw.edu.pl

Abstract

It is known that provability in propositional intuitionistic logic is Pspace-complete. As Pspace is closed under complements, there must exist a Logspace-reduction from refutability to provability. We describe a direct translation below: given a formula φ, we define $\overline{\varphi}$ so that $\overline{\varphi}$ is provable iff φ is not.

Preliminaries

We consider propositional formulas built from the connectives $\wedge$, $\vee$, $\to$ and $\bot$. Negation $\neg\alpha$ is defined as $\alpha \to \bot$. We assume that implication is right-associative, i.e., we write $\alpha \to \beta \to \gamma$ for $\alpha \to (\beta \to \gamma)$. If $\mathcal{S} = \{\alpha_1, \ldots, \alpha_k\}$ then $\mathcal{S} \to \beta$ abbreviates any formula of the form $\alpha_1 \to \cdots \to \alpha_k \to \beta$ (disregarding the order of premises). Variables and $\bot$ are *atoms*.

We use an indexed version of cut-free sequent calculus to derive judgments of the form $\Gamma \vdash_t \alpha$, where Γ is a set of formulas, α is a formula, and t is a natural number. The meaning of $\vdash_t$ is that the depth of the proof does not exceed t.

$$\Gamma, \alpha \vdash_t \alpha \ (\mathrm{Ax}) \qquad \Gamma, \bot \vdash_t \alpha \ (\mathrm{L}\bot)$$

$$\frac{\Gamma, \gamma, \delta, \gamma \wedge \delta \vdash_t \alpha}{\Gamma, \gamma \wedge \delta \vdash_{t+1} \alpha}\,(\mathrm{L}\wedge) \qquad \frac{\Gamma \vdash_t \gamma \quad \Gamma \vdash_t \delta}{\Gamma \vdash_{t+1} \gamma \wedge \delta}\,(\mathrm{R}\wedge)$$

$$\frac{\Gamma, \gamma, \gamma \vee \delta \vdash_t \alpha \quad \Gamma, \delta, \gamma \vee \delta \vdash_t \alpha}{\Gamma, \gamma \vee \delta \vdash_{t+1} \alpha}\,(\mathrm{L}\vee) \qquad \frac{\Gamma \vdash_t \gamma}{\Gamma \vdash_{t+1} \gamma \vee \delta}\,(\mathrm{R}\vee)\frac{\Gamma \vdash_t \delta}{\Gamma \vdash_{t+1} \gamma \vee \delta}$$

$$\frac{\Gamma, \gamma \to \delta \vdash_t \gamma \quad \Gamma, \gamma \to \delta, \delta \vdash_t \alpha}{\Gamma, \gamma \to \delta \vdash_{t+1} \alpha}\,(\mathrm{L}\to) \qquad \frac{\Gamma, \gamma \vdash_t \delta}{\Gamma \vdash_{t+1} \gamma \to \delta}\,(\mathrm{R}\to)$$

The system above is sound and complete: a judgment $\Gamma \vdash \alpha$ is provable in the standard sense iff $\Gamma \vdash_{n^2} \alpha$, where n is the number of all subformulas occurring in Γ, α.

The construction

In what follows we fix a formula φ and we define a formula $\overline{\varphi}$ to satisfy the equivalence:

$$\vdash \overline{\varphi} \quad \Leftrightarrow \quad \nvdash \varphi. \tag{*}$$

Assume that φ is of length n and let $\mathcal{S}$ be the set of all subformulas of φ. Then $\mathcal{S}$ has at most n elements. Define $N = n^2$ and $T = \{0, \ldots, N\}$.

For every $\alpha, \beta \in \mathcal{S}$, and every $t \in T$, the following propositional symbols occur in $\overline{\varphi}$:

– $D_{\alpha,t}$ – "Disprove α in t steps"
– $A_{\alpha,t}$ – "Assumption α not added at time t"
– $X_{\alpha,t}$ – "Axiom not able to derive α at time t"
– $R_{\alpha,t}$ – "Right rule not able to derive α at time t"
– $L_{\alpha,\beta,t}$ – "Left rule for β not able to derive α at time t"

Let $\mathcal{A}_t = \{A_{\alpha,t} \mid \alpha \in \mathcal{S}\}$, $\mathcal{A}_t\backslash\beta = \{A_{\alpha,t} \mid \alpha \in \mathcal{S} \wedge \alpha \neq \beta\}$, $\mathcal{A}_t\backslash\beta\gamma = \{A_{\alpha,t} \mid \alpha \in \mathcal{S} \wedge \alpha \neq \beta, \gamma\}$. Then define $\mathcal{A}_{\alpha,t\uparrow} = \{A_{\alpha,u} \mid u \geq t\}$ and $\mathcal{A}_{\Delta,t\uparrow} = \bigcup\{\mathcal{A}_{\alpha,t\uparrow} \mid \alpha \notin \Delta\}$, when $\Delta \subseteq \mathcal{S}$.

The formula $\overline{\varphi}$ has the form $\Gamma \to D_{\varphi,N}$, where Γ is the set consisting of the implicational formulas listed below. First, all formulas of the form $\mathcal{A}_{\alpha,0\uparrow} \cup \mathcal{A}_{\perp,0\uparrow} \to D_{\alpha,0}$, as well as all the atoms $A_{\alpha,N}$ belong to Γ. Then, for $t > 0$:

- For every $\alpha \in \mathcal{S}$, we have $\mathcal{M}_{\alpha,t} \to D_{\alpha,t} \in \Gamma$, where $\mathcal{M}_{\alpha,t}$ is the set of the following propositional variables:
  - $X_{\alpha,t}$;
  - $L_{\alpha,\beta,t}$, for any non-variable $\beta \in \mathcal{S}$, including the case $\beta = \perp$;
  - $R_{\alpha,t}$, in case α is not an atom.
- Formulas $\mathcal{A}_{\alpha,t\uparrow} \to X_{\alpha,t}$ belong to Γ.
- If $\alpha = \gamma \wedge \delta$ then $(\mathcal{A}_{t-1} \to D_{\gamma,t-1}) \to R_{\alpha,t}$ and $(\mathcal{A}_{t-1} \to D_{\delta,t-1}) \to R_{\alpha,t}$ belong to Γ.
- If $\alpha = \gamma \vee \delta$ then $(\mathcal{A}_{t-1} \to D_{\gamma,t-1}) \to (\mathcal{A}_{t-1} \to D_{\delta,t-1}) \to R_{\alpha,t}$ is in Γ.
- If $\alpha = \gamma \to \delta$ then $(\mathcal{A}_{t-1}\backslash\gamma \to D_{\delta,t-1}) \to R_{\alpha,t}$ belongs to Γ.
- Every formula $\mathcal{A}_{\beta,t\uparrow} \to L_{\alpha,\beta,t}$ is in Γ.
- If $\beta = \gamma \wedge \delta$ then $(\mathcal{A}_{t-1}\backslash\gamma\delta \to D_{\alpha,t-1}) \to L_{\alpha,\beta,t}$ is in Γ.
- If $\beta = \gamma \vee \delta$ then $(\mathcal{A}_{t-1}\backslash\gamma \to D_{\alpha,t-1}) \to L_{\alpha,\beta,t}$, and $(\mathcal{A}_{t-1}\backslash\delta \to D_{\alpha,t-1}) \to L_{\alpha,\beta,t}$ are in Γ.
- If $\beta = \gamma \to \delta$ then $(\mathcal{A}_{t-1} \to D_{\gamma,t-1}) \to L_{\alpha,\beta,t}$ and $(\mathcal{A}_{t-1}\backslash\delta \to D_{\alpha,t-1}) \to L_{\alpha,\beta,t}$ are in Γ.

Lemma 1. *For every $t \in T$, every $\Delta \subseteq \mathcal{S}$, and every $\alpha \in \mathcal{S}$:*

$$\Delta \nvdash_t \alpha \quad \Longleftrightarrow \quad \Gamma, \mathcal{A}_{\Delta,t\uparrow} \vdash D_{\alpha,t}.$$

The main equivalence (*) follows for $\Delta = \varnothing$, $\alpha = \varphi$, and $t = N$. (Note that $\mathcal{A}_{\varnothing,N\uparrow} \subseteq \Gamma$.)

Comments:
– Works like [2, 1] introduce systems of rules to derive refutability. We propose an alternative: rather than introducing new rules, we apply the old ones to a different task.
– The construction uses an explicit counting from 0 to n^2. It can be reduced down to n: we will do it in the final version of this paper.
– Provability in IPC can be represented by monotone alternating automata, essentially implementing the games of [3]. The above construction can therefore be simplified as complementation for such automata.
– The formula $\overline{\varphi}$ (and any simple encoding of a monotone automaton) uses only implication and is of order (depth) at most 3. This gives yet another argument that for every formula φ, there is a formula ψ of order 3 such that φ is provable iff so is ψ. The formula ψ is not equivalent to φ, but is computable in logarithmic space (an analogy to Cnf-Sat).
– Having the decision problem reduced in Logspace to simple formulas, one can consider various simplifications/heuristics (like joining and deleting some components), working towards an intuitionistic analogue of Davis-Putnam algorithm.

References

[1] Camillo Fiorentini and Mauro Ferrari. A forward unprovability calculus for intuitionistic propositional logic. In R. A. Schmidt and C. Nalon, editors, *Proc. TABLEAUX 2017*, volume 10501 of *LNAI*, pages 114–130. Springer, 2017.

[2] Luis Pinto and Roy Dyckhoff. Loop-free construction of counter-models for intuitionistic propositional logic. In R. Behara, M. Fritsch and R.G. Lintz, editors, *Symposia Gaussiana, Conf. A, 1993*, pages 225—-232. De Gruyter, 1995.

[3] Paweł Urzyczyn. Intuitionistic games: Determinacy, completeness, and normalization. *Studia Logica*, 104(5):957–1001, 2016.

7 Formalizing mathematics with types (WG4)

Mechanized Undecidability Results for Propositional Calculi

Andrej Dudenhefner

Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
dudenhefner@ps.uni-saarland.de

Introduction The line of work on decidability of propositional, possibly sub-intuitionistic, Hilbert-style calculi was motivated by Tarski in 1946. Historically, the most prominent results include undecidability theorems by Linial and Post [11] published in 1949. Subsequently, a variety of problems regarding propositional calculi (for an overview see [17]) was studied. A well-known positive result is that provability in intuitionistic propositional implicational logic, i.e. the calculus having as axioms $a \to b \to a$ and $(a \to b \to c) \to (a \to b) \to a \to c$, is PSPACE-complete [15]. Complementarily, Singletary [14] proved in 1974 that there exists a purely implicational propositional calculus which can represent any r.e. degree.

Modern results [5, 6] show that problems of recognizing axiomatizations and completeness are undecidable for propositional calculi containing the axiom $a \to b \to a$. Such calculi can be understood as simply typed [9, 3] combinatory logics [2, 10] exposing an undecidable type inhabitation problem. Notably, modern results sometimes include a mechanization of the main argument (in case of [6] using the Lean [12] proof assistant).

This work gives fully constructive, mechanized (in the Coq proof assistant) synthetic many-one reductions from the Turing machine halting problem (Halt) to the following two problems. First, provability of a given proposition in a fixed implicational propositional calculus. Second, provability of a fixed proposition in a given implicational propositional calculus.

Preliminaries *Propositions* s, t are defined as $s, t \in \mathbb{F} ::= a \mid s \to t$, where $a \in \mathbb{A}$ ranges over atoms. A *substitution* $\zeta : \mathbb{A} \to \mathbb{F}$ is lifted to propositions by $\zeta(s \to t) = \zeta(s) \to \zeta(t)$. An *environment* Γ is a finite set of propositions. A proposition t is *derivable* from Γ, if there is a proposition s in Γ and substitution ζ such that $t = \zeta(s)$, or there is a proposition s such that both propositions s and $s \to t$ are derivable from Γ. Clearly, derivability of a proposition s in Γ corresponds to provability of s in a Hilbert-style calculus having propositions in Γ as axioms.

This work revisits undecidability of the following two problems.

Problem 1. Let Γ be an environment. Given a proposition t, is t derivable from Γ?

Problem 2. Let t be a proposition. Given an environment Γ, is t derivable from Γ?

The main contribution improves upon existing work in the following two aspects.

- The given argument is fully constructive.
- The given argument is mechanized (in the Coq proof assistant) as part of a uniform framework of computational reductions in Coq [8, 7].

Relying on the infrastructure of the existing framework provides an advantage. For it suffices to reduce an existing, more convenient problem (called a *seed* in the framework) to a target problem in order to obtain a comprehensive reduction from Halt to the target problem.

The relevant Coq development can be found at [1] and is currently being integrated into [8], containing a suitable seed. The above preliminaries are mechanized in `HSC/HSC_prelim.v`. Derivability is mechanized as the inductive predicate `hsc (Gamma: list formula) : formula -> Prop`.

Synthetic Many-one Reductions The notion of synthetic reducibility [7] is based on synthetic computability theory [4]. The key insight is that in the convenient language of constructive type theory, proofs are computable functions that can be used to both describe and verify reductions between decision problems. This avoids an explicit construction of Turing machines, which are formally required for such reductions. Synthetic many-one reducibility of a predicate `p : X -> Prop` to a predicate `q : Y -> Prop` is mechanized in `Reduction.v` as

```
Definition reduces X Y (p : X -> Prop) (q : Y -> Prop) :=
  exists f : X -> Y, forall x, p x <-> q (f x).
Notation "p ⪯ q" := (reduces p q) (at level 50).
```

Fundamentally, `Halt ⪯ q` describes the undecidability of `q`.

Derivability from a Fixed Environment Akin to the result by Singletary [14], we explicitly give an environment Γ (mechanized as `ΓPCP` in `BMPCP_to_HSC_PRV.v`) such that it is undecidable, whether a given proposition s is derivable from Γ (mechanized as `HSC_PRV ΓPCP : formula -> Prop`). Concretely, we reduce the binary modified Post correspondence problem [13] (BMPCP), a seed mechanized as `BMPCP` in `PCP/PCP.v`, to derivability in Γ. The particular synthetic many-one reduction is mechanized in `BMPCP_to_HSC_PRV.v` as

```
Theorem BMPCP_to_HSC_PRV : BMPCP ⪯ HSC_PRV ΓPCP.
```

The key idea is to encode *state* transition, where state is a pair $((Q, P), (x, y))$ such that P is the list of word pairs given by the BMPCP instance, Q is a suffix of P, and x, y are constructed by respective repeated concatenation of words from P. The fixed environment Γ consists of encodings of state transitions in accordance with the search for a solution of an arbitrary instance of BMPCP. Overall, a BMPCP instance $((v, w), P)$ is solvable iff the proposition encoding the state $((((v, w)+P), ((v, w)+P)), (v, w))$ is derivable from Γ. Notably, axioms in Γ are derivable from the single axiom $a \to b \to a$.

From the existing infrastructure of the framework we obtain the following result for free

```
Theorem HSC_PRV_undec : Halt ⪯ HSC_PRV ΓPCP.
```

Derivability of a Fixed Proposition Considering the environment as problem input, sometimes called *relativized* provability, is related to the problem of recognizing axiomatizations of a fixed theory. Therefore, as a starting point, we use the mechanization of the main argument for undecidability of recognizing axiomatizations of a calculus having $a \to b \to a$ as its only axiom [6]. In particular, whether from a given environment Γ the fixed proposition $a \to b \to a$ is derivable (mechanized as `HSC_AX` in `HSC/HSC_AX.v`). The key idea, again, is to encode solvability of BMPCP such that both state transition and input is encoded in the environment (for details see [6]). Notably, the constructed environment contains only axioms derivable from $a \to b \to a$.

Traditional argumentation [11, 16, 5, 6] and its mechanization relies on the (non-computable) principle of excluded middle. However, in the setting of synthetic reductions, it is not obvious to what extent the use of non-computable principles is legitimate. This work provides a constructive reformulation, similar to continuation passing, of the development from [6]. Surprisingly, the code size of the resulting Coq development is only half that of the original Lean development. This is partly due to a more concise `ssreflect`-style proofs and the `lia` tactic for integer arithmetic. The result is mechanized in `BMPCP_to_HSC_AX.v` as

```
Theorem BMPCP_to_HSC_AX : BMPCP ⪯ HSC_AX.
```

Again, the framework readily provides

```
Theorem HSC_AX_undec : Halt ⪯ HSC_AX.
```

References

[1] Andrej Dudenhefner. Mechanized Reductions from the Binary Modified Post Correspondence Problem to Problems for Propositional Calculi. https://github.com/uds-psl/2020-types-propositional-calculi. Accessed: 2020-02-03.

[2] Hendrik P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, 2nd Edition. Elsevier Science Publishers, 1984.

[3] Hendrik P. Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic, Cambridge University Press, 2013.

[4] Andrej Bauer. First Steps in Synthetic Computability Theory. *Electr. Notes Theor. Comput. Sci.*, 155:5–31, 2006.

[5] Grigoriy V. Bokov. Undecidable problems for propositional calculi with implication. *Logic Journal of the IGPL*, 24(5):792–806, 2016.

[6] Andrej Dudenhefner and Jakob Rehof. Lower End of the Linial-Post Spectrum. In Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, editors, *23rd International Conference on Types for Proofs and Programs, TYPES 2017, May 29-June 1, 2017, Budapest, Hungary*, volume 104 of *LIPIcs*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[7] Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-Related Computational Reductions in Coq. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 253–269, 2018.

[8] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq Library of Undecidable Problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020.

[9] J. Roger Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science, vol. 42, Cambridge University Press, 2008.

[10] J. Roger Hindley and Jonathan P. Seldin. *Lambda-calculus and Combinators, an Introduction*. Cambridge University Press, 2008.

[11] Samuel Linial and Emil L. Post. Recursive Unsolvability of the Deducibility, Tarski's Completeness and Independence of Axioms Problems of Propositional Calculus. *Bulletin of the American Mathematical Society*, 55:50, 1949.

[12] Microsoft Research. The Lean Proof Assistant. https://leanprover.github.io/. Accessed: 2020-01-07.

[13] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[14] Wilson E. Singletary. Many-one Degrees Associated with Partial Propositional Calculi. *Notre Dame Journal of Formal Logic*, XV(2):335–343, 1974.

[15] Richard Statman. Intuitionistic Propositional Logic Is Polynomial-space Complete. *Theoretical Computer Science*, 9:67–72, 1979.

[16] Mary K. Yntema. A detailed argument for the Post-Linial theorems. *Notre Dame Journal of Formal Logic*, 5(1):37–50, 1964.

[17] Evgeny Zolin. Undecidability of the Problem of Recognizing Axiomatizations of Superintuitionistic Propositional Calculi. *Studia Logica*, 102(5):1021–1039, 2014.

Towards Extraction of Continuity Moduli in Coq

Yannick Forster[1], Dominik Kirst[1], and Florian Steinberg[2]

[1] Saarland University
Saarland Informatics Campus, Saarbrücken, Germany
{forster,kirst}@ps.uni-saarland.de
[2] INRIA Saclay
Paris, Framce
fsteinberg@gmail.com

Abstract

We report on a work-in-progress extraction of continuity information for Coq functionals on Baire space, i.e. of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$. The extraction is implemented as a MetaCoq plugin and generates a certified modulus function, given a term in the System T fragment of Coq. In fact, the extraction first reifies Coq definitions into a syntactic representation of System T and subsequently employs a constructive and informative continuity theorem for System T following Escardó.

It is a well-studied property of constructive mathematics that the functions definable in a purely constructive setting à la Bishop are computable. As a consequence, definable functions over Baire space into natural numbers are continuous.

The latter has been established explicitly for various phrasings of constructive mathematics, for instance Gödel's System T, expressing the primitive recursive functions of finite types. For more profound constructive foundations, such as the dependent type theory underlying the Coq proof assistant, it is clear that at least T-definable functions can be shown continuous in the system itself. We provide a first step towards exploiting such continuity information by implementing a plugin for automatically extracting the modulus of continuity of T-definable Coq functionals of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$.

Such a functional f on Baire space is called *continuous* if it only accesses finitely many positions of every input sequence $\alpha : \mathbb{N} \to \mathbb{N}$, i.e. if there is a function $\mu_f : (\mathbb{N} \to \mathbb{N}) \to \mathcal{L}(\mathbb{N})$ such that for every α it holds that $f\,\alpha = f\,\beta$ for every β agreeing with α on the positions listed in $\mu_f\,\alpha$. The function μ_f is called the *modulus of continuity* of f and can be extracted for all T-definable Coq functionals by first reifying into a syntactic representation of System T (Section 2) using the MetaCoq framework [3] and then executing a constructive and informative continuity proof for System T (Section 1) as implemented by Escardó in Agda [1]. The Coq code is available at https://www.ps.uni-saarland.de/extras/modulus-extraction/.

1 Extracting Continuity Moduli from System T

We follow Escardó's Agda development [1] to implement a Coq procedure that computes and verifies the modulus of continuity for T-definable functionals, i.e. functionals that are the denotation of a term of System T. Using standard techniques from programming language semantics, Escardó gives a compact mechanisation that straightforwardly translates to Coq. As intended for the calculus of constructions at the core of Coq's type theory, most of the logical statements from the Agda proof can be placed in the (impredicative) propositional universe $\mathbb{P}$ while only the definition of continuity remains in the (predicative) computational hierarchy $\mathbb{T}$, so that the modulus function can be extracted. Moreover, as in Escardó's proof, we rely on an intrinsically typed Church-style representation of System T, i.e. do not model untyped syntax.

2 A Modulus Extraction Plugin

We utilise MetaCoq to reify Coq's System T fragment syntactically into an inductive type representing untyped System T syntax, reminiscent of reification into the untyped λ-calculus [2]. MetaCoq provides an inductive type `Ast.term` mirroring the OCaml datatype used to implement Coq and a monad `TemplateMonad` which can be used to access effects like unfolding of names, quoting a Coq term into its `Ast.term` representation or unquoting an `Ast.term` representation back into a Coq term.

Our plugin thus first calls a monadic program `Reify` translating from `Ast.term` into System T. Monadic programs can be executed using a vernacular command. To execute `Reify`, a user can type `MetaCoq Run (Reify r f)` to reify `f` into System T automatically and add the result to the environment as definition named `r`. The reification function is essentially the (partial) identity, just renaming constructors of Coq (e.g. Coq's application `Ast.tApp`) into constructors of a type `SystemT.term` representing untyped System T (e.g. `SystemT.app`). The two datatypes are displayed below, the alignment hints at how the translation works:

```
Module Ast.
  Inductive term : Set :=
  | tRel : nat -> term
  | tConstruct : inductive -> nat ->
      universe_instance -> term
  | tFix : mfixpoint term -> nat -> term
  | tLambda : name -> term -> term -> term
  | tApp : term -> term -> term
  (* ... *).
End Ast.
```

```
Module SystemT.
  Inductive term : Type :=
  | var : nat -> term
  | zero : term
  | succ : term
  | rec : type -> term
  | lambda : type -> term -> term
  | app : term -> term -> term.

End SystemT.
```

As a second step, we translate the untyped System T representation to the intrinsically typed representation for the continuity proof by calling a certified type inference procedure for T. In the last step, we utilise the continuity theorem for System T from above to implement a plugin function called `ExtractModulus`. Given a Coq term f of the expected type, it reifies the functional into System T, infers typing information, employs the continuity theorem, and checks that the denotation of the System T representation is indeed the initial functional. The plugin can be called as `MetaCoq Run (ExtractModulus mod f)` where the modulus of `f` is saved as the definition `mod` together with a proof that it indeed is the modulus of continuity.

3 Future Directions

We see the current implementation as ground for further investigations in the extraction of continuity information in Coq. In the current state, the plugin has hardly any practical applications. To make it useful in applications like computable real analysis [5] we want to extend to more base types in System T like $\mathbb{B}$, sums, pairs, lists, or rational numbers.

Furthermore, Coq functions are mostly defined using a match/fix representation of recursion instead of a explicit eliminators, which our reification cannot yet deal with.

MetaCoq allows users to verify plugins in principle. For our plugin, this would mainly be a verification of the reification function, which would be eased by relying on the verified type inference function for Coq [4].

Lastly, we would like to investigate the continuity notion and proofs for T-functions of higher type as in [6] and for larger fragments of Coq including dependent and informative types.

References

[1] M. Escardó. Continuity of Gödel's System T definable functionals via effectful forcing. *Electronic Notes in Theoretical Computer Science*, 298:119–141, 2013.

[2] Y. Forster and F. Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[3] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter. The MetaCoq Project. June 2019.

[4] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):8, 2020.

[5] F. Steinberg, L. Théry, and H. Thies. Quantitative Continuity and Computable Analysis in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[6] C. Xu. A syntactic approach to continuity of T-definable functionals, 2019.

Constructing Higher Inductive Types as Groupoid Quotients

Niels van der Weide

Radboud University, Nijmegen, The Netherlands
nweide@cs.ru.nl

The Martin-Löf identity type, also known as *propositional equality*, represents provable equality in type theory [9]. This type is defined polymorphically over all types and has a single introduction rule representing reflexivity. The eliminator, often called the J-rule or path induction, is used to prove symmetry and transitivity. Note that in particular, we can talk about the identity type of an already established identity type. This can be iterated to obtain an infinite tower of types, which has the structure of an ∞-groupoid [8, 12].

The J-rule is also the starting point of *homotopy type theory* [11]. In that setting, types are seen as spaces, inhabitants are seen as points, proofs of identity are seen as paths, and paths between paths are seen as homotopies. In mathematical terms, type theory can be interpreted using simplicial sets [6]. In the resulting model, not every inhabitant of the identity type is equal to reflexivity.

Assuming the univalence axiom, we can construct types for which we can prove that not every two inhabitants of the identity type are equal. One example is the universe [11] while other examples can be obtained by using *higher inductive types* (HITs).

Higher inductive types (HITs) generalize inductive types by allowing constructors for paths, paths between paths, and so on. While inductive types are specified by giving the arities of the operations [3], for higher inductive types one must also specify the arities of the paths, paths between paths, and so on. The resulting higher inductive type is freely generated by the provided constructors. To make this concrete, let us look at some examples [11]:

$$
\begin{array}{l}
\texttt{Inductive}\ S^1 := \\
\mid\ \mathbf{base}_{S^1} : S^1 \\
\mid\ \mathbf{loop}_{S^1} : \mathbf{base}_{S^1} = \mathbf{base}_{S^1}
\end{array}
$$

$$
\begin{array}{l}
\texttt{Inductive}\ \mathcal{T}^2 := \\
\mid\ \mathbf{base} : \mathcal{T}^2 \\
\mid\ \mathbf{loop_l}, \mathbf{loop_r} : \mathbf{base} = \mathbf{base} \\
\mid\ \mathbf{surf} : \mathbf{loop_l} \bullet \mathbf{loop_r} = \mathbf{loop_r} \bullet \mathbf{loop_l}
\end{array}
$$

The first one, S^1, represents the circle. It is generated by a point constructor $\mathbf{base}_{S^1} : S^1$ and a path constructor $\mathbf{loop}_{S^1} : \mathbf{base}_{S^1} = \mathbf{base}_{S^1}$. The second one, $\mathcal{T}^2$, represents the torus. This type is generated by a point constructor $\mathbf{base}$, two path constructors $\mathbf{loop_l}$ and $\mathbf{loop_r}$ of type $\mathbf{base} = \mathbf{base}$, and a homotopy constructor $\mathbf{surf} : \mathbf{loop_l} \bullet \mathbf{loop_r} = \mathbf{loop_r} \bullet \mathbf{loop_l}$ where $p \bullet q$ denotes the concatenation of p and q. Note that constructors depend on previously given constructors in the specification. For both types, introduction, elimination, and computation rules can be given [11].

In this talk, we study a schema of higher inductive types that allows defining types by giving constructors for the points, paths, and homotopies. All of these constructors can be recursive, but they can only have a finite number of recursive arguments. To guarantee that all constructors are finitary, we use finitary polynomials. Note that recursion is necessary to cover

examples such as W-types, the set truncation, algebraic theories, and the integers. A similar scheme was studied by Dybjer and Moeneclaey and they interpret HITs on this scheme in the groupoid model [4]. Kaposi and Kovács study a more general scheme of HITs [5].

Say that a type X is *1-truncated* if for all $x, y : X$, $p, q : x = y$, and $r, s : p = q$ we have $r = s$, and a *1-type* is a type which is 1-truncated. An example of a 1-type is the circle [7], which we mentioned before. Groupoids are related to 1-types via the *groupoid quotient* [10], which takes a groupoid G and returns 1-type whose points are objects of G and whose paths are morphisms in G. Note that the type of univalent groupoids is equivalent to the type of 1-types [2].

The goal of this talk is to show that finitary 1-truncated higher inductive types can be derived from simpler principles. More specifically, every finitary 1-truncated HIT can be constructed in a type theory with propositional truncations, set quotients, and groupoid quotients. The result of this talk can be used to simplify the semantic study of finitary 1-truncated HITs. Instead of verifying the existence of a wide class of HITs, one only needs to check the existence of propositional truncations and groupoid quotients.

For the proof, we use the following approach

1. We define within type theory an internal definition of signatures for HITs. These signatures allow path and homotopy constructors.

2. For each signature, we define bicategories of algebras in both 1-types and groupoids.

3. Then we define displayed algebras and the induction principle for HITs. Note that the induction principle only allows mapping to 1-types since we study 1-truncated HITs. We follow that up by showing that biinitial algebras in 1-types satisfy the induction principle.

4. After that, we construct a biadjunction between the bicategories of algebras in 1-types and algebras in groupoids.

5. We finish the proof by constructing the biinitial algebra in groupoids. From this, we get that every signature has a HIT.

To construct the desired biadjunction, we make use of the displayed machinery introduced by Ahrens *et al.* [1] and for this reason, we define the bicategories of algebras using displayed bicategories. We also give the notion of *displayed biadjunction* and show that each displayed biadjunction gives rise to a biadjunction between the total bicategories.

Formalization All results in this talk are formalized in Coq using the UniMath library [13]. The formalization can be found here:

`https://github.com/nmvdw/GrpdHITs`

Acknowledgments The author thanks Herman Geuvers, Dan Frumin, Niccolò Veltri, Benedikt Ahrens, and Ali Caglayan for helpful comments and discussions.

References

[1] Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccoló Veltri, and Niels van der Weide. Bicategories in univalent foundations. *arXiv preprint arXiv:1903.01152v3v2*, 2020.

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

[3] Peter Dybjer. Inductive Families. *Formal aspects of computing*, 6(4):440–465, 1994.

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

[5] Ambrus Kaposi and András Kovács. A Syntax for Higher Inductive-Inductive Types. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 20:1–20:18, 2018.

[6] Chris Kapulkin and Peter LeFanu Lumsdaine. The Simplicial Model of Univalent Foundations (after Voevodsky). *Journal of the European Mathematical Society*, 2018.

[7] Daniel R. Licata and Michael Shulman. Calculating the Fundamental Group of the Circle in Homotopy Type Theory. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 223–232, 2013.

[8] Peter LeFanu Lumsdaine. Weak ω-categories from Intensional Type Theory. In *International Conference on Typed Lambda Calculi and Applications*, pages 172–187. Springer, 2009.

[9] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[10] Kristina Sojakova. Higher Inductive Types as Homotopy-Initial Algebras, 2016.

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

[12] Benno van den Berg and Richard Garner. Types are Weak ω-Groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

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

Modular Confluence for Rewrite Rules in MetaCoq

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

[1] TU Delft, Delft, Netherlands
[2] Gallinette Project-Team, Inria, Nantes, France

Dependently typed languages provide strong guarantees of correctness for our programs and proofs, but they can be hard to use and extend. To increase their practicability and expressivity, they can be extended with user-defined *rewrite rules* [2]. For example, rewrite rules allow us to define 'parallel' plus that reduces on both sides. It is defined by one symbol `pplus` and the following rules:

$$\begin{array}{ll} m : \mathbb{N} \vdash \texttt{pplus}\ 0\ m \hookrightarrow m & n, m : \mathbb{N} \vdash \texttt{pplus}\ (\texttt{S}\ n)\ m \hookrightarrow \texttt{S}\ (\texttt{pplus}\ n\ m) \\ n : \mathbb{N} \vdash \texttt{pplus}\ n\ 0 \hookrightarrow n & n, m : \mathbb{N} \vdash \texttt{pplus}\ n\ (\texttt{S}\ m) \hookrightarrow \texttt{S}\ (\texttt{pplus}\ n\ m) \end{array}$$

Since rewrite rules are applied to expressions appearing at the type level, they interact directly with the type system. Hence they can very easily break expected good properties of these systems, *e.g.,* termination, confluence, canonicity or subject reduction [3]. Among those, confluence is a key ingredient to retain subject reduction and as such is essential.

We present a new criterion to ensure confluence of rewrite rules – and thus subject reduction of the system – for the predicative calculus of cumulative inductive constructions (PCUIC) as formalised in MetaCoq [6]. This criterion is *modular*: adding new rewrite rules that satisfy the criterion again yields a confluent system without having to check new properties of pre-existing rewrite rules. We have formalized this criterion in MetaCoq and extended the existing proof of confluence. The proof however is not particular to the system and should adapt easily to other settings such as Agda, allowing us to extend the Agda implementation [2] with a confluence checker.

Rewrite rules in our setting. We extend PCUIC with blocks consisting of symbol declarations and rewrite rules where the head of each rule is one of the locally defined symbols. A rewrite rule is given as $\Delta \vdash l \hookrightarrow r$ where:

- Δ is the telescope of pattern variables, which should be the only free variables in l and r, all pattern variables appear *linearly* in l (*i.e.,* exactly one time);
- l is the elimination of some symbol declared in the same block, where eliminations include application to a pattern, projection from a record type, and pattern-matching where the return predicate and branches are also patterns;
- patterns can be pattern variables applied to locally bound variables, bound variables, or λ-abstractions where the type and body are both patterns;
- r is unconstrained (except its free variables).

In order to preserve subject reduction, we also ask that there exists a type A such that $\Delta \vdash l : A$ and $\Delta \vdash r : A$. Rewrite rules are interpreted as follows (σ instantiates the pattern variables):

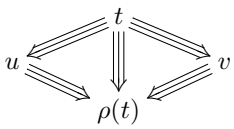
$$\frac{\sigma : \Gamma \to \Delta}{l\sigma \longrightarrow r\sigma}$$

The Tait–Martin-Löf criterion. This method to prove confluence [5] has been used in MetaCoq to show the confluence of PCUIC [7]. Basically, it requires to introduce—beside the standard reduction ($\longrightarrow$)—a notion of parallel reduction ($\Rightarrow$) which *may* do one-step reduction in all its subterms and such that:

$$\longrightarrow \subseteq \Rightarrow \subseteq \longrightarrow^\star$$

so that confluence of parallel reduction is sufficient to get confluence of reduction. In particular, parallel reduction is reflexive.

Confluence of $\Rightarrow$ relies on the existence of an auxiliary function ρ such that $\rho(t)$ is the best parallel reduct of t, *i.e.*, whenever $t \Rightarrow u$ we also have $u \Rightarrow \rho(t)$. This means that $t \Rightarrow \rho(t)$ holds as well. The existence of such a function ρ is called the *triangle property* and allows us to derive confluence by glueing two triangles as illustrated below:



Modular confluence for rewrite rules. For a given set S of rewrite rules to be confluent with the rest of the system, we ask that it satisfies the triangle property *locally* in the following sense. Assuming that the set S of rewrite rules is ordered (for instance, by their order of appearance), we define the function ρ_S by applying the first rule in S that matches and applying ρ_S recursively on the pattern variables. For instance, for the rule $x, y : A \vdash F\ (G\ x)\ y \hookrightarrow H\ x\ y$, the definition of ρ_S is $\rho_S(F\ (G\ x)\ y) = H\ (\rho_S\ x)\ (\rho_S\ y)$. Then, the local triangle property for S asks that ρ_S satisfies the triangle property for the rules in S.

Assuming this local triangle property for S and that the theory satisfies the triangle property (with function ρ), we can show that the theory extended with the new set S of rewrite rules still satisfies the triangle property, and thus is confluent. The new auxiliary function ρ is just obtained merging ρ_S into ρ; replacing every recursive call to ρ_S by a recursive call to ρ.

One key ingredient in the proof is the fact that if a term matches one of the rules in S, it still matches this rule after applying any of the 'old' rules to (subterms of) this term. This is very important for the modular reasoning to go through, and it is not satisfied by *non-linear* rewrite rules. Indeed, when a non-linear rule matches, this match is easily broken by rewriting only one of the two occurrences of the non-linear pattern, whatever this reduction is. Thus, the confluence of non-linear rewrite rules cannot be analyzed this way.

Coming back to the example of parallel plus, the local triangle property is satisfied after adding the following (admissible) reduction rule, with high priority:

$$n, m : \mathbb{N} \vdash \texttt{pplus}\ (\texttt{S}\ n)\ (\texttt{S}\ m) \Rightarrow \texttt{S}\ (\texttt{S}\ (\texttt{pplus}\ n\ m))$$

Extended this way, our criterion shows that adding `pplus` to the theory does not break confluence.

Compared to the literature on higher-order rewriting [4, 1, 8], our criterion may look very restrictive. The reason is that we focus on modularity and formal provability, and we work with PCUIC rather than a simpler Pure Type System as is usually the case in the literature.

Formalisation. The formalisation of this modular approach to confluence can be found at `https://github.com/TheoWinterhalter/template-coq/tree/rewrite-rules`. It is a fork of the MetaCoq repository where we add rewrite rules to the global environment. The confluence proof is in the process of being updated accordingly although there are still remaining assumptions at the time of submission.

References

[1] Frédéric Blanqui, Claude Kirchner, and Colin Riba. On the confluence of lambda-calculus with conditional rewriting. *Theor. Comput. Sci.*, 411(37):3301–3327, 2010.

[2] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In *Types for Proofs and Programs*, TYPES, 2016.

[3] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. How to tame your rewrite rules. In *Types for Proofs and Programs*, TYPES, 2019.

[4] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical computer science*, 192(1):3–29, 1998.

[5] Gert Smolka. Confluence and normalization in reduction systems. Lecture Notes, 2015.

[6] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. To appear in Journal of Automated Reasoning, 2020.

[7] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *PACMPL*, 4(POPL), 2020.

[8] Vincent van Oostrom. Confluence by decreasing diagrams. In *Rewriting Techniques and Applications, RTA 2008*, volume 5117 of *Lecture Notes in Computer Science*, 2008.

8 Types and verification (WG4)

Program Analysis via Monadic Translations

Nils Köpp[1], Thomas Powell[2], and Chuangjie Xu[1]

[1] Ludwig-Maximilians-Universität München, Munich, Germany
[2] Technische Universität Darmstadt, Darmstadt, Germany

Abstract

We introduce a parametrized monadic translation of Gödel's System T (and its extensions), and prove a corresponding fundamental theorem of logical relation. By instantiating the monad and the base case of the logical relation, we reveal various properties and structures of programs in T such as majorizability and (uniform) continuity.

Introduction Our goal is to reveal information that is implicit in the structure of functional programs via syntactic translations. Inspired by [vdB19], we introduce a notion of nucleus to parametrize the translation. A nucleus looks like a monad in the form of a Kleisli extension, but it is not required to satisfy the monad laws. Our translated programs remain in the same language instead of some monadic metalanguage as in [Uus02, Pow19], so that the targeting information of a program, such as certain bounds or moduli, can be represented as a program in the same language. Note that our translation structurally corresponds to Gentzen's negative translation [Ish00], whereas those in [Uus02, Pow19] correspond to Kolmogorov's and Kuroda's. For simplicity, we firstly work with Gödel's System T and then demonstrate how our method is generalized to the extension T with (co)inductive types.

System T and J-translation Recall that the term language of System T can be given by

$$\begin{array}{ll} \text{Type} & \sigma, \tau ::= \mathbb{N} \mid \sigma \to \tau \\ \text{Term} & t, u ::= x \mid \lambda x.t \mid tu \mid 0 \mid \mathsf{suc} \mid \mathsf{rec} \end{array}$$

with the usual typing and reduction rules (specifically, we take $\mathsf{rec} : \sigma \to (\mathbb{N} \to \sigma \to \sigma) \to \mathbb{N} \to \sigma$). A *nucleus* relative to T is a triple $(\mathsf{J}\mathbb{N}, \eta, \kappa)$ where $\mathsf{J}\mathbb{N}$ is a type and

$$\eta : \mathbb{N} \to \mathsf{J}\mathbb{N} \qquad \kappa : (\mathbb{N} \to \mathsf{J}\mathbb{N}) \to \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$$

are terms in T. We may write g^κ instead of $\kappa(g)$ for $g : \mathbb{N} \to \mathsf{J}\mathbb{N}$. Given a nucleus $(\mathsf{J}\mathbb{N}, \eta, \kappa)$, we can translate T into itself as follows: For each type ρ of T, assign a type ρ^J by

$$\begin{aligned} \mathbb{N}^\mathsf{J} &:= \mathsf{J}\mathbb{N} \\ (\sigma \to \tau)^\mathsf{J} &:= \sigma^\mathsf{J} \to \tau^\mathsf{J}. \end{aligned}$$

Given a term $t : \rho$ of T and a mapping of variables $x : \tau$ in its context to variables $x^\mathsf{J} : \tau^\mathsf{J}$, assign a term $t^\mathsf{J} : \rho^\mathsf{J}$ by induction on t as follows

$$\begin{array}{lll} (x)^\mathsf{J} := x^\mathsf{J} & (\lambda x.t)^\mathsf{J} := \lambda x^\mathsf{J}.t^\mathsf{J} & (tu)^\mathsf{J} := t^\mathsf{J} u^\mathsf{J} \\ 0^\mathsf{J} := \eta 0 & \mathsf{suc}^\mathsf{J} := \kappa(\eta \circ \mathsf{suc}) & \mathsf{rec}^\mathsf{J} := \lambda a f.\mathrm{ke}(\mathsf{rec}(a, f \circ \eta)) \end{array}$$

where $\mathrm{ke}_\tau : (\mathbb{N} \to \tau^\mathsf{J}) \to \mathsf{J}\mathbb{N} \to \tau^\mathsf{J}$ is the extension of κ to arbitrary type τ of T which is defined inductively on τ. Our main result is the following variant of the fundamental theorem of logical relation [Sta85]:

Theorem 1. *Let* $(\mathsf{J}\mathbb{N}, \eta, \kappa)$ *be a nucleus. Given a binary relation* $\mathrm{R}_\mathbb{N} \subseteq \mathbb{N} \times \mathsf{J}\mathbb{N}$*, we extend it to* $\mathrm{R}_\rho \subseteq \rho \times \rho^\mathsf{J}$ *for arbitrary type* ρ *of* T *by defining*

$$f\ \mathrm{R}_{\sigma\to\tau}\ g \coloneqq \forall x^\sigma, a^{\sigma^\mathsf{J}}\,(x\ \mathrm{R}_\sigma\ a \to fx\ \mathrm{R}_\tau\ ga)\,.$$

If $\mathrm{R}_\mathbb{N}$ *satisfies*

$$\forall n\,(n\ \mathrm{R}_\mathbb{N}\ \eta n) \qquad \text{and} \qquad \forall i\,(fi\ \mathrm{R}_\mathbb{N}\ gi) \to f\ \mathrm{R}_{\mathbb{N}\to\mathbb{N}}\ g^\kappa \tag{$\dagger$}$$

then $t\ \mathrm{R}_\rho\ t^\mathsf{J}$ *for any closed term* $t : \rho$ *of* T*.*

Instantiations Our first example is Howard's majorizability relation [How73] which extends the usual ordering $\leq$ on natural numbers to functionals of arbitrary finite type in the same way as in the above theorem, *i.e.*

$$n \lhd_\mathbb{N} m \coloneqq n \leq m$$
$$f \lhd_{\sigma\to\tau} g \coloneqq \forall x, y\,(x \lhd_\sigma y \to fx \lhd_\tau gy)\,.$$

We say t is *majorized* by u if $t \lhd u$, and call u a *majorant* of t. Majorizability plays an important role in models of higher-order calculi and more recently in the proof mining program [Koh08]. Howard shows that each closed term of T is majorized by some closed term of T. This result fits perfectly into our framework: Taking $\mathsf{J}\mathbb{N} = \mathbb{N}$ and defining $\eta : \mathbb{N} \to \mathbb{N}$ and $g^\kappa : \mathbb{N} \to \mathbb{N}$ by

$$\eta(n) \coloneqq n \qquad \begin{aligned} g^\kappa(0) &\coloneqq g(0) \\ g^\kappa(n+1) &\coloneqq \max(g^\kappa(n), g(n+1)) \end{aligned}$$

for any $g : \mathbb{N} \to \mathbb{N}$, we can easily show that $\lhd_\mathbb{N}$ fulfills the conditions ($\dagger$), and thus have $t \lhd t^\mathsf{J}$ for every closed term t of T by Theorem 1. In this instantiation, the translation is an algorithm to construct majorants, and its correctness is given by the fundamental theorem of logical relation. Working with other nuclei, we can *e.g.* compute moduli of (uniform) continuity [Xu19b] and construct bar-recursion functionals [OS18] for T-definable functions via the translation. More details and examples can be found in our preprint and Agda development [Xu19a].

Extensions Now consider the extension of T with products and sums. Products are translated component-wise. For sums, we generalize the notion of nucleus to an endo-map J on types with terms η, κ of suitable types. Then the type translation is extended with

$$(\sigma \times \rho)^\mathsf{J} \coloneqq \sigma^\mathsf{J} \times \rho^\mathsf{J} \qquad (\sigma + \rho)^\mathsf{J} \coloneqq \mathsf{J}\left(\sigma^\mathsf{J} + \rho^\mathsf{J}\right)$$

corresponding to Gentzen's negative translation of conjunction and disjunction. We can further extend our framework to certain *inductive* and *co-inductive* types. As an example consider the type $\mathrm{S}(\tau)$ of infinite streams with constants

$$\mathrm{hd} : \mathrm{S}(\tau) \to \tau \qquad \mathrm{tl} : \mathrm{S}(\tau) \to \mathrm{S}(\tau) \qquad \mathrm{coit} : (\sigma \to \tau) \to (\sigma \to \sigma) \to \sigma \to \mathrm{S}(\tau).$$

To translate $\mathrm{S}(\tau)$, we recursively translate its argument τ and then apply the map J, *i.e.*

$$(\mathrm{S}(\tau))^\mathsf{J} \coloneqq \mathsf{J}(\mathrm{S}(\tau^\mathsf{J})).$$

The constants of $\mathrm{S}(\tau)$ are translated as follows:

$$\mathrm{hd}^\mathsf{J} \coloneqq \mathrm{ke}\,(\mathrm{hd}) : \mathsf{J}(\mathrm{S}(\tau^\mathsf{J})) \to \tau^\mathsf{J} \qquad \mathrm{tl}^\mathsf{J} \coloneqq \kappa\,(\eta \circ \mathrm{tl}) : \mathsf{J}(\mathrm{S}(\tau^\mathsf{J})) \to \mathsf{J}(\mathrm{S}(\tau^\mathsf{J}))$$
$$\mathrm{coit}^\mathsf{J} \coloneqq \lambda h, t, x.\ \eta\,(\mathrm{coit}(h, t, x)) : (\sigma^\mathsf{J} \to \tau^\mathsf{J}) \to (\sigma^\mathsf{J} \to \sigma^\mathsf{J}) \to \sigma^\mathsf{J} \to \mathsf{J}(\mathrm{S}(\tau^\mathsf{J})).$$

As an application consider the nucleus $\mathsf{J}(\mathrm{S}(\tau)) \coloneqq (\mathbb{N} \to \tau)$. This will translate programs with streams as above into programs using function-representations of streams instead.

References

[How73] William A. Howard. Hereditarily majorizable functionals of finite type. In *Metamathematical investigation of intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, pages 454–461. Springer, Berlin, Heidelberg, 1973.

[Ish00] Hajime Ishihara. A note on the Gödel-Gentzen translation. *Mathematical Logic Quarterly*, 46(1):135–137, 2000.

[Koh08] Ulrich Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics*. Springer Monographs in Mathematics, 2008.

[OS18] Paulo Oliva and Silvia Steila. A direct proof of Schwichtenberg's bar recursion closure theorem. *The Journal of Symbolic Logic*, 83(1):70–83, 2018.

[Pow19] Thomas Powell. A unifying framework for continuity and complexity in higher types, 2019. arXiv:1906.10719 [cs.LO].

[Sta85] Richard Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65:85–97, 1985.

[Uus02] Tarmo Uustalu. Monad translating inductive and coinductive types. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002). Lecture Notes in Computer Science, vol 2646*, pages 299–315. Springer, 2002.

[vdB19] Benno van den Berg. A Kuroda-style j-translation. *Archive for Mathematical Logic*, 58(5–6):627–634, 2019.

[Xu19a] Chuangjie Xu. A Gentzen-style monadic translation of Gödel's System T. arXiv:1908.05979 [cs.LO]. Agda development available at http://cj-xu.github.io/agda/ModTrans/index.html, 2019.

[Xu19b] Chuangjie Xu. A syntactic approach to continuity of T-definable functionals, 2019. To appear at *Logical Methods in Computer Science*. arXiv:1904.09794 [math.LO].

Contextual Modal Types for Algebraic Effects and Handlers

Nikita Zyuzin and Aleksandar Nanevski

IMDEA Software Institute

Algebraic effects and handlers [2, 9] provide a modular and compositional description for computational effects. In this view, *only* a designated set of operations invokes side effects during evaluation of a term. Moreover, the use of such operations can be eliminated by a handler that provides definitions for these operations.

In this work, we propose that algebraic effects and handlers can be naturally typed using a variation of Contextual Modal Type Theory [5]. CMTT distinguishes between contextual modal variables $u :: A[\Delta]$, and normal variables $x : A$, having two separate contexts Θ and Γ for them respectively. The type $[\Delta]A$ classifies terms that depend on normal variables in the context Δ, but no other normal variables. The typing rules of CMTT are:

$$\Box I\,\frac{\Theta; \Delta \vdash s : A}{\Theta; \Gamma \vdash \texttt{box}\ \Delta.\ s : [A]\Delta} \qquad \Box E\,\frac{\Theta; \Gamma \vdash s : [A]\Delta \qquad \Theta, u :: A[\Delta]; \Gamma \vdash t : B}{\Theta; \Gamma \vdash \texttt{let box}\ u = s\ \texttt{in}\ t : B}$$

$$\text{ctxhyp}\,\frac{(u :: A[\Delta]) \in \Theta \qquad \Theta; \Gamma \vdash \sigma : \Delta}{\Theta; \Gamma \vdash \texttt{handle}\ u\ \texttt{with}\ \sigma : A} \qquad \text{explsub}\,\frac{\Theta; \Gamma \vdash s_i : A_i \qquad i = \overline{1, n}}{\Theta; \Gamma \vdash \langle s_i / x_i, \dots \rangle : (x_i : A_i, \dots)}$$

The $\Box I$ rule introduces contextual modality; the term under the box constructor may *only* use normal variables bound by the context Δ, but not by Γ. The rules $\Box E$ and `ctxhyp` allow to use a boxed term, by binding it to a variable in the context ($\Box E$), and then using this variable (`ctxhyp`). One can use contextual variables by applying an explicit substitution (`explsub`) that replaces all `box`-bound variables with appropriate terms.

We propose to view the context Δ as the algebraic theory of computations of type $[\Delta]A$. Thus the calculus immediately provides a type-and-effect system. Consider the program P that uses algebraic effects from theories $\texttt{St} \mathrel{\widehat{=}} \mathit{get} : \mathit{unit} \to \mathit{nat}, \mathit{put} : \mathit{nat} \to \mathit{unit}$ of state and $\texttt{Ex} \mathrel{\widehat{=}} \mathit{raise} : \mathit{unit} \to \bot$ of exceptions:

```
P ≙ box St, Ex. let x = get() in
                  if x = 42 then raise() else put(x + 1)
```

P has the type $[\texttt{St}, \texttt{Ex}]\texttt{unit}$, signifying that P can cause effects from `St` and `Ex`, but no others. To run P, we must provide an explicit substitution σ that defines all the operations from `St` and `Ex`, and then execute `let box u = P in handle u with` σ`.`

Explicit substitution is thus similar to handling of algebraic effects, which is why we write `handle` to denote applying it. Unfortunately, the similarity is not strong enough, as in CMTT we cannot define a handler neither for state nor for exceptions in this example. The problem arises because we want to use the operations *get*, *set* and *raise* as generic effects[8] and pass no continuation arguments that would allow state manipulation in substitution clauses.

In this paper we modify CMTT to adapt its notion of handling to programming with algebraic effects, preserving the type-and-effect discipline of contextual modal types. Specifically: (1) We use contextual modal types to denote algebraic theories of effectful computations; (2) We adopt the judgment for monadic computation $e \div A$ from [6], whose terms make the sequencing of effects explicit. Only terms of this judgment can be boxed; (3) When used left of $\vdash$ to declare variables, we generalize the judgment to $c \div A \Rightarrow B$. The generalized judgement

denotes computations *hypothetical* in A (thus, it classifies effectful functions), and gives us suitable typing for the effect operators; (4) We extend the notion of applying explicit substitution in CMTT to handling of algebraic effects. Most important typing rules of our system are:

$$\text{cnthyp}\frac{(k \approx A \otimes B \rightsquigarrow C) \in \Gamma \quad \Theta;\Gamma \vdash s : A \quad \Theta;\Gamma \vdash t : B}{\Theta;\Gamma \vdash \texttt{throw}\ k\ s\ t \div C} \qquad \text{ophyp}\frac{\Theta;\Gamma, op \div A \Rightarrow B \vdash s : A}{\Theta;\Gamma, op \div A \Rightarrow B \vdash op\ s \div B}$$

$$\text{handler}\frac{\Theta;\Gamma, z : D, x : A_i, k \approx B_i \otimes D \rightsquigarrow C \vdash e_i \div C \qquad i = \overline{1,n}}{\Theta;\Gamma \vdash z : D.\{op_i(x,k) \Rightarrow e_i, \dots\} \div D.[op_i \div A_i \Rightarrow B_i, \dots] \triangleright C}$$

$$\text{ctxhyp}\frac{(u :: A[\Delta]) \in \Theta \qquad \Theta;\Gamma \vdash h \div B.[\Delta] \triangleright C \qquad \Theta;\Gamma \vdash t : B \qquad \Theta;\Gamma, x : A, z' : B \vdash e \div C}{\Theta;\Gamma \vdash \texttt{handle}\ u\ \texttt{with}\ h\ \texttt{from}\ t\ \texttt{to}\ x.z'.e \div C}$$

In more detail, in $e \div A$, e is an effectful computation that runs and produces a value of type A. This judgement forces computations to be written as a sequence of `let` forms. The judgments $e \div A$ and $c \div A \Rightarrow B$ are related by the `effhyp` rule. Rule `conthyp` provides use for *continuation* variables also typed by a hypothetical judgement.

We use the rules $\Box I$ and $\Box E$ from CMTT, adapting them to the judgement for computations. The `handler` rule specifies handling of each operation $op_i \div A_i \Rightarrow B_i$, with corresponding terms e_i. We type check these terms in the extended context, where $z : D$ is a handler-bound variable shared by all operations, x is the operation's op_i parameter, and $k \approx B_i \otimes D \rightsquigarrow C$ is the continuation for op_i. The continuation takes op_i's output and a new value for the shared variable, returning a value of the return type of the handler.

Finally, the `ctxhyp` rule types handling of computations, and subsumes the old rule from CMTT. Here, handler h serves as the explicit substitution: h substitutes all the free variables from the algebra Δ. `from` t specifies initial value t for the handler-bound variable. `to` $x.z'.e$ binds the final value and shared variable after handling to respectively x and z' in e.

With our new typing rules, we can handle the program P, now setting $\texttt{St} \mathrel{\widehat{=}} get \div unit \Rightarrow nat, put \div nat \Rightarrow unit$ and $\texttt{Ex} \mathrel{\widehat{=}} raise \div unit \Rightarrow \bot$:

```
let box u = P in handle u with
  s: nat. { get(v, k) => throw k s s,
            put(v, k) => throw k v (),
            raise(v, k) => ret ((), s) }
from 0 to x. s'. ret (x, s')
```

This handler has the type $nat.[get \div unit \Rightarrow nat, put \div nat \Rightarrow unit, raise \div unit \Rightarrow \bot] \triangleright unit \times nat$. The handler-bound variable s preserves the state of the program between different invocations of *get* and *put*: the body of an operation receives current state in the context and specifies the resulting state when calling continuation. `from` 0 sets the initial state to 0 for the handled computation. Finally, we return the resulting value and state as a pair.

We are currently working on proving type soundness of the proposed calculus. In future, we plan to scale to dependent types, as context with dependent types will allow us to concisely specify algebraic theories *with equations between algebraic operations*. As CMTT has already been shown to support dependent types [5], we expect that our extension to algebraic effects will support them as well. Dependent types will also facilitate verification of programs with generic effects, and we plan to explore potential connections with separation logic. These will provide a different verification perspective, compared to other systems with algebraic effects and dependent types, e.g. [1]. We also plan to explore abstraction over handlers and contexts in the contextual types [4], to obtain abstraction over algebraic theories as an alternative to row polymorphism [3]. Additionally, our use of contextual types gives another perspective on scoping for algebraic effects and handlers [7, 10].

References

[1] Danel Ahman. Handling fibred algebraic effects. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[2] Andrej Bauer. What is algebraic about algebraic effects and handlers? *arXiv preprint arXiv:1807.05923*, 2018.

[3] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages*, 3(POPL):6, 2019.

[4] Andrew Cave and Brigitte Pientka. First-class substitutions in contextual type theory. In *Logical Frameworks & Meta-languages: Theory & Practice (LFMTP)*, pages 15–24, 2013.

[5] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):23, 2008.

[6] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(4):511–540, 2001.

[7] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 809–818, 2018.

[8] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied categorical structures*, 11(1):69–94, 2003.

[9] Matija Pretnar and Gordon D Plotkin. Handling algebraic effects. *Logical Methods in Computer Science*, 9, 2013.

[10] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 1–12, 2014.

Project Proposal: Relieving User Effort for the Auto Tactic in Coq with Machine Learning

Lasse Blaauwbroek*

Czech Institute for Informatics, Robotics and Cybernetics, Czech Republic
Radboud University Nijmegen, the Netherlands
lasse@blaauwbroek.eu

We propose to enhance the `auto` tactic in Coq with machine learning, aiming to reduce the effort the user has to put in designing hint databases for `auto`. We seek ideas and advice regarding the specific type of ML that would be appropriate in this context.

Proof Styles in Coq The Coq Proof Assistant supports many methods of proving a theorem. One can either directly write proof terms, or choose one of the tactical languages like Ltac [2] or Mtac [3]. Then there are custom sets of tactics for Ltac like SSReflect [5]. However, even within one of these paradigms there are still different styles of proving available. Some people advocate structured proofs using Coq's built-in bullet points, writing every step of the proof explicitly in hopes of increasing readability. Other people try to write very compact and tailored tactic scripts that prove a lemma in one step. This usually results in shorter and easier to maintain proofs, often at the cost of readability. All of these styles have their place, depending on the mathematical domain one is trying to formalize.

In this proposal we will focus on one specific proving style described and popularized by Adam Chlipala [1]. The concept is to provide as little proof information as possible within the tactic script of a lemma. Usually this means that one critical step of the proof is explicitly stated in the script, while the rest of the proof is pieced together by automation. For example, the critical step can be to use induction on a specific variable of the lemma. The resulting cases of the induction principle then have to be solved by the built-in `auto` tactic. This tactic is a generic prover that uses hints previously provided by the user to guide proof search. These hints usually consist of a recipe on how to use a previously declared sublemma of the proof. However, it is also possible to teach the `auto` tactic how and when to use custom tactics and complete decision procedures.

This approach has two main advantages. (1) It keeps the actual proof scripts short and therefore maintainable. If something in the development changes it should be easy to go through the development and fix the hints and proofs, if necessary at all. (2) By only using the `auto` tactic the user is forced to tease out important information about the proof and refactor this into a lemma, hint or tactical procedure. In this way, all the crucial steps will be explicitly declared and can be easily understood by readers without bogging them down with the straightforward details of the proof. The truth of a lemma would ideally be evident to a reader simply by thinking about previously provided hints for a bit, just like it is to Coq.

The `auto` Tactic Fundamentally, the auto tactic is a simple search procedure. For a proof state it can compile a list of possible actions to take, together with a priority for these actions. The resulting search space is traversed in BFS or DFS fashion until either a full proof is found or a limit is reached. The interesting part is that the list of possible actions is compiled from so-called hint-databases. These databases are meant to contain usage information for lemmas and tactics in the current development. Users can add and remove information from a database on the fly by using variations of the `Hint` vernacular. We give some examples that add hints to a database.

- `Hint Resolve thmx` will tell `auto` to try and unify the current goal with the conclusion of theorem `thmx`. On success, auto will replace the current goal with the assumptions of `thmx`.

*This work was supported by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466)

- Let `thmy` be a theorem that has an equality as its conclusion. `Hint Rewrite -> thmy` will tell auto to rewrite the goal using `thmy` if the goal unifies with the left-hand side of the equality. Any possible assumptions of `thmy` are added to the proof state.
- `Hint Extern tac` can be used to register any tactic `tac` to be run by auto. This can be useful for making `auto` do things like normalize terms or other simple steps that never go wrong. Also, since this vernacular gives us access to the full power of the tactical language, it allows us to encode much more complicated hints, as we will elaborate below.

Hint databases have to be designed with great care. Adding the wrong lemmas to a database can lead to a very large search space and even infinite loops. The larger and more complicated a development is, the more problematic this becomes. To reduce the branching factor in such developments, a hint can have a gate specifying the conditions that need to be met before the hint is used. In its simplest form, this can be a pattern that must be matched to the goal before the hint applies. It is, however, possible to write arbitrarily complicated gates using Ltac as a programming language. This way, the hint can be accepted or rejected based on the full contents of the proof state. Philosophically speaking, the goal of writing a gate is to capture the domain specific knowledge and intuition that the user has on how and when to use a lemma or tactic. A simple example is a gate for the lemma $a < b \to b < c \to a < c$. We want to apply this lemma to a goal $x < z$ only if we can expect to find a suitable y. Therefore, the gate will be a pattern on the proof state: `?x < ?y,...,?y < ?z` $\vdash$ `?x < ?z`. Note that this gate is very strict, and a much more complicated one might be required in practice.

Experience tells us that for a decently sized development, the branching factor of the search performed by `auto` has to be kept well below 1.5 to keep the system usable.[1] The gating required to reach this can be quite laborious. Conversely, it tends to not be very difficult to achieve a branching factor smaller than five. Our proposal is to bridge the gap between these factors using machine learning, bringing together the best of human intuition and the computers ability to do the grunt work.

Machine Learning for `auto` Our proposal to incorporate machine learning into auto consists of gathering information on previous runs of the auto tactic. The idea is that at the beginning of a development, hints and proofs are usually much simpler, allowing auto to find a proof easily. We can then record which hints ended up being fruitful in the context of which proof state. As the development progresses, the system can then start to leverage this information to prioritize the list of available actions to `auto` in a proof state. Actions will be more important if they have been used previously in similar states. The amount of actions the machine learning has to choose from will be quite limited because the gating of the user has already weeded out most inapplicable actions.

One fundamental challenge is that the system will not have a lot of data to learn from. This is because within a development a hint associated with a lemma would normally be used tens or at most hundreds of times. The system needs to learn quickly in terms of data. On the other hand, because there will be very few choices to be made at each point, there will be quite a lot of time to consider each choice. For these reasons, most traditional learning techniques, like neural networks, will not be immediately applicable. The simplest approach is to extract features from proof states, and perform a direct comparison with previous states. However, more symbolic methods such as approximate substring matching between goal and lemma may also be applicable [4]. During TYPES we would like to gather feedback about other techniques that may suit this setting.

[1] This is partially due to the fact that Coq users generally do not, can not, and often do not want to replace the auto tactic with the found solution like is common in Isabelle. Therefore, to get a good experience, the search has to be completed within seconds.

References

[1] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[2] David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[3] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: typed tactics for backward reasoning in coq. *PACMPL*, 2(ICFP):78:1–78:31, 2018.

[4] Jiaying Wang, Xiaochun Yang, Bin Wang, and Chengfei Liu. An adaptive approach of approximate substring matching. In *Database Systems for Advanced Applications - 21st International Conference, DASFAA 2016, Dallas, TX, USA, April 16-19, 2016, Proceedings, Part I*, pages 501–516, 2016.

[5] Iain Whiteside, David Aspinall, and Gudmund Grov. An essence of ssreflect. In *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, pages 186–201, 2012.

Type-preserving compilation via dependently typed syntax

Andreas Abel*

Department of Computer Science and Engineering, Gothenburg University

The *CompCert* project [Leroy, 2009] produced a verified compiler for a large fragment of the C programming language. The CompCert compiler is implemented in the type-theoretic proof assistant Coq [INRIA, 2019], and is fully verified: there is a proof that the semantics of the source program matches the semantics of the target program. However, full verification comes with a price: the majority of the formalization is concerned not with the runnable code of the compiler, but with properties of its components and proofs of these properties. If we are *not* willing to pay the price of full verification, can we nevertheless profit from the technology of type-theoretic proof assistants to make our compilers *safer* and *less likely* to contain bugs?

In this talk, I am presenting a compiler for a small fragment of the C language using *dependently-typed syntax* [Benton et al., 2012, Allais et al., 2018]. A typical compiler is proceeding in phases: parsing, type checking, code generation, and finally, object/binary file creation. Parsing and type checking make up the *front end*, which may report syntax and type errors to the user; the other phases constitute the *back end* that should only fail in exceptional cases. After type checking succeeded, we have to deal only with well-typed source programs, whose abstract syntax trees can be captured with the indexed data types of dependently-typed proof assistants and programming languages like Agda [Agda developers, 2019], Coq, Idris [Brady, 2013], Lean [de Moura et al., 2015] etc. More concisely, we shall by *dependently-typed syntax* refer to the technique of capturing well-typedness invariants of syntax trees.

Representing also typed assembly language [Morrisett et al., 1999] via dependently-typed syntax, we can write a type-preserving compiler whose type soundness is given *by construction*. In the talk, the target of compilation is a fragment of the Java Virtual Machine (JVM) enriched by some *administrative instructions* that declare the types of local variables. With JVM being a stack machine, instructions are indexed not only by the types of the local variables, but also by the types of the stack entries before and after the instruction. However for instructions that change the control flow, such as unconditional and conditional jumps, we need an additional structure to ensure type safety. Jumps are safe if the jump target has the same machine typing than the jump source. By *machine typing* we mean the pair of the types of the local variables and the types of the stack entries. Consequently, each label (i. e., jump target) needs to be assigned a machine type and can only be targeted from a program point with the same machine type. Technically, we represent labels as machine-typed de Bruijn indices, and control-flow instructions are indexed by a context of label types. We then distinguish two types of labels:

1. Join points, e. g., labels of statements following an `if-else` statement. Join points can be represented by a `let` binding in the abstract JVM syntax.

2. Looping points, e. g., labels at the beginning of a `while` statement that allow back jumps to iterate the loop. Those are represented by `fix` (recursion).

Using dependently-typed machine syntax, we ensure that *well-typed jumps do not miss*. As a result, we obtain a type-preserving compiler by construction, with a good chance of full correctness, since many compiler faults already break typing invariants. Intrinsic well-typedness also allows us to write the compiler as a total function from well-typed source to typed assembly, and totality can be automatically verified by the Agda type and termination checker.

*Supported by VR grants 621-2014-4864 and 2019-04216 and EU Cost Action CA15123.

References

Agda developers. *Agda 2.6.0 documentation*, 2019. http://agda.readthedocs.io/en/v2.6.0/.

G. Allais, R. Atkey, J. Chapman, C. McBride, and J. McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proceedings of the ACM on Programming Languages*, 2(ICFP):90:1–90:30, 2018. https://doi.org/10.1145/3236785.

N. Benton, C.-K. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012. https://doi.org/10.1007/s10817-011-9219-0.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. http://dx.doi.org/10.1017/S095679681300018X.

L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, vol. 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. https://doi.org/10.1007/978-3-319-21401-6_26.

INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.9 edition, 2019. http://coq.inria.fr/.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. http://doi.acm.org/10.1145/1538788.1538814.

J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999. https://doi.org/10.1145/319301.319345.

9 Types and computation

Multi Types for Strong Call-by-Value

Beniamino Accattoli[1], Andrea Condoluci[2], Giulio Guerrieri[3], Maico Leberle[1], and Claudio Sacerdoti Coen[2]

[1] LIX, Inria, Palaiseau, France
beniamino.accattoli@inria.fr maico-carlos.leberle@inria.fr
[2] Department of Computer Science, University of Bologna, Bologna, Italy
andreacondoluci@gmail.com sacerdot@cs.unibo.it
[3] Department of Computer Science, Universtity of Bath, Bath, United Kingdom
g.guerrieri@bath.ac.uk

Plotkin's call-by-value λ-calculus [31] is at the heart of programming languages such as OCaml and proof assistants such as Coq. In the study of programming languages, call-by-value (CbV) evaluation is usually *weak* (it does not reduce under abstractions) and terms are assumed to be *closed*. These constraints give rise to an elegant framework, called *Closed CbV* by Accattoli and Guerrieri [4].

It often happens, however, that one needs to go beyond the perfect setting of Closed CbV by considering *Strong CbV* (reduction under abstractions is allowed and terms may be open), or the intermediate setting of *Open CbV* (evaluation is weak but terms may be open). The need arises, most notably, when describing the implementation model of Coq, as done by Grégoire and Leroy [19], but also from other motivations, such as denotational semantics [30, 33, 6, 10], monad and CPS translations [26, 34, 35, 16, 21], bisimulations [23], partial evaluation [22], linear logic proof nets [1], cost models [7].

Naïve Extension of CbV. In call-by-name (CbN) turning to open terms or strong evaluation is harmless because CbN does not impose any special form to the arguments of β-redexes. On the contrary, turning to Open or Strong CbV is delicate. While some fundamental properties such as confluence and standardization hold also in such cases, as showed by Plotkin's himself, others break as soon as one considers open terms. As pointed out by Paolini and Ronchi Della Rocca [30, 28, 33], denotational semantics that are *adequate*[1] for Closed CbV are no longer adequate for the extended settings. Roughly, there are terms that are semantically divergent, that is, with empty semantics, while they are normal forms with respect to Plotkin's rules, and should have non-empty semantics.

The discrepancy can be seen from a logical point of view. These terms diverge also if seen as (recursively typed) linear logic proof nets, as pointed out by Accattoli [1], or as terms in the computational interpretation of sequent calculus due to Curien and Herbelin [11]. One may even trace the problems of Closed CbV to Plotkin's seminal paper, where he points out an asymmetry between CbN and CbV with respect to CPS translations. This fact led to a number of studies [26, 34, 35, 24, 16, 21] that introduced many proposals of improved calculi for CbV.

A Survey of the Theory of Open CbV. Recently, Accattoli and Guerrieri provided an in-depth study of Open CbV, providing operational [4] and semantic [5] analyses, connected at the quantitative level.

[1] A model is *adequate* when the interpretation, or semantics, of a term is non-degenerated—typically non-empty—iff the term normalizes. For a CbV model, adequacy is somewhat mandatory, because any model of CbN provides a non-adequate model of CbV that does not model the CbV behavior.

Operationally, they show that various proposed extensions of Closed CbV are termination equivalent[2] in the open setting, and evaluation in these calculi takes the same number of β-steps. Namely, they relate Accattoli and Paolini's *value substitution calculus* [6], which is a term syntax for the proof-nets representation of λ-terms according to the CbV translation to linear logic, Paolini and Della Rocca's *fireball calculus*—used to design abstract machines by Grégoire and Leroy [19] and Accattoli and Sacerdoti Coen [7]—and the *value sequent calculus*, that is, the intuitionistic and CbV sub-calculus of Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [11].

Semantically, they show that multi types (aka non-idempotent intersection types, aka relational semantics) in their CbV declination due to Ehrhard [17] characterize termination in the previous calculi, and induce an adequate semantics, thus solving the inadequacy issue of the naïve extension. In particular, such semantics is deeply related to linear logic, see for instance [18, 8, 20, 13], and its CbN variant is the base of the main quantitative models of the λ-calculus, see [32, 12, 14, 17, 9, 29, 27, 25, 3]. Accattoli and Guerrieri further use multi types derivations to provide exact bounds to β-steps to normal form and for the size of normal forms, along the lines of de Carvalho's work for CbN multi types [12, 14].

Strong CbV. Here we extend the described theory of Open CbV to *strong* evaluation, at the operational and semantic levels. We characterize termination in the value substitution calculus, and use a new strong evaluation strategy with the diamond property. We also extract the number of its steps and the size of the normal form from type derivations, in an exact way. Multi types are also used for our main operational result, the normalization theorem for the strategy, by exploiting an elegant approach used in de Carvalho et al. [15] and Mazza et al. [25]. Our study preserves the deep connection with linear logic and its quantitative semantics, and mimics the proof technique used by Accattoli, Graham-Lengrand, and Kesner [3].

Let us clarify a basic point. The strong setting is subtler, and extending the results from Closed and Open CbV to Strong CbV is non-trivial. Closed CbV can be thought as a *call-by-normal-form* calculus: β-redexes can be fired only when arguments are values (*i.e.* variables or abstractions), and values are normal forms. A similar property can be recovered also in Open CbV [4]. In Strong CbV such an essence is lost. To give an idea, if Ω denotes the usual diverging term then in Strong CbV $\lambda x.\Omega$ diverges (evaluation under abstraction) while $(\lambda y.z)(\lambda x.\Omega)$ may both diverge and reduce to z—the argument needs not be fully normalized, but only reduced to a (weak) value. This is mandatory, to be conservative over Closed and Open CbV.

As a consequence, even though we use Ehrhard's multi types system for CbV [17] and Accattoli and Paolini's value substitution calculus [6], we have to introduce some operational and semantic refinements. Operationally, we define an evaluation strategy for the value substitution calculus that we shall show to be normalizing. Its role is analogous to the leftmost-outermost strategy of the λ-calculus. A notable difference, however, is that the strategy is itself non-deterministic, but in a harmless way, as it is diamond.

Semantically, to characterize the set of normalizing terms via multi types, we need to consider type derivations where occurrences of the empty multiset are restricted via a notion of polarity. Only for these type derivations the size decreases after each step of the evaluation strategy, allowing us to extract not only qualitative but also quantitative information.

This work is extracted from a more comprehensive paper [2] where we study Strong CbV not only from an operational and semantic viewpoint, but also at the implementative level, via an innovative study of abstract machines that do not break the fragile adequate relationship with the semantics.

[2] Two calculi X and Y are termination equivalent if t terminates in X iff t terminates in Y.

References

[1] Beniamino Accattoli. Proof nets and the call-by-value λ-calculus. *Theor. Comput. Sci.*, 606:2–24, 2015.

[2] Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, Maico Leberle, and Claudio Sacerdoti Coen. Strong call-by-value. Submitted to LICS 2020.

[3] Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds. *PACMPL*, 2(ICFP):94:1–94:30, 2018.

[4] Beniamino Accattoli and Giulio Guerrieri. Open Call-by-Value. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016*, volume 10017 of *Lecture Notes in Computer Science*, pages 206–226. Springer, 2016.

[5] Beniamino Accattoli and Giulio Guerrieri. Types of fireballs. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, pages 45–66, 2018.

[6] Beniamino Accattoli and Luca Paolini. Call-by-Value Solvability, revisited. In *FLOPS*, pages 4–16, 2012.

[7] Beniamino Accattoli and Claudio Sacerdoti Coen. On the Relative Usefulness of Fireballs. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015*, pages 141–155. IEEE Computer Society, 2015.

[8] Antonio Bucciarelli and Thomas Ehrhard. On phase semantics and denotational semantics: the exponentials. *Ann. Pure Appl. Logic*, 109(3):205–241, 2001.

[9] Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. A relational semantics for parallelism and non-determinism in a functional setting. *Ann. Pure Appl. Logic*, 163(7):918–934, 2012.

[10] Alberto Carraro and Giulio Guerrieri. A Semantical and Operational Account of Call-by-Value Solvability. In *FOSSACS 2014*, pages 103–118, 2014.

[11] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000.

[12] Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul.* Thèse de doctorat, Université Aix-Marseille II, 2007.

[13] Daniel de Carvalho. The relational model is injective for multiplicative exponential linear logic. In *CSL 2016*, pages 41:1–41:19, 2016.

[14] Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Math. Str. in Comput. Sci.*, 28(7):1169–1203, 2018.

[15] Daniel de Carvalho, Michele Pagani, and Lorenzo Tortora de Falco. A semantic measure of the execution time in linear logic. *Theor. Comput. Sci.*, 412(20):1884–1902, 2011.

[16] Roy Dyckhoff and Stéphane Lengrand. Call-by-Value lambda-calculus and LJQ. *J. Log. Comput.*, 17(6):1109–1134, 2007.

[17] Thomas Ehrhard. Collapsing non-idempotent intersection types. In *CSL*, pages 259–273, 2012.

[18] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[19] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP '02*, pages 235–246. ACM, 2002.

[20] Giulio Guerrieri, Luc Pellissier, and Lorenzo Tortora de Falco. Computing Connected Proof(-Structure)s from their Taylor Expansion. In *FSCD 2016*, pages 20:1–20:18, 2016.

[21] Hugo Herbelin and Stéphane Zimmermann. An operational account of Call-by-Value Minimal and Classical λ-calculus in Natural Deduction form. In *TLCA*, pages 142–156, 2009.

[22] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[23] Søren B. Lassen. Eager Normal Form Bisimulation. In *20th IEEE Symposium on Logic in Computer Scienc, LICS 2005*, pages 345–354. IEEE Computer Society, 2005.

[24] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, Call-by-value, Call-by-need and the Linear λ-Calculus. *TCS*, 228(1-2):175–210, 1999.

[25] Damiano Mazza, Luc Pellissier, and Pierre Vial. Polyadic approximations, fibrations and intersection types. *PACMPL*, 2:6:1–6:28, 2018.

[26] Eugenio Moggi. Computational λ-Calculus and Monads. In *LICS '89*, pages 14–23, 1989.

[27] C.-H. Luke Ong. Quantitative semantics of the lambda calculus: Some generalisations of the relational model. In *LICS 2017*, pages 1–12, 2017.

[28] Luca Paolini. Call-by-Value Separability and Computability. In *ICTCS*, pages 74–89, 2002.

[29] Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Essential and relational models. *Mathematical Structures in Computer Science*, 27(5):626–650, 2017.

[30] Luca Paolini and Simona Ronchi Della Rocca. Call-by-value Solvability. *ITA*, 33(6):507–534, 1999.

[31] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[32] Alberto Pravato, Simona Ronchi Della Rocca, and Luca Roversi. The call-by-value λ-calculus: a semantic investigation. *Math. Str. in Comput. Sci.*, 9(5):617–650, 1999.

[33] Simona Ronchi Della Rocca and Luca Paolini. *The Parametric λ-Calculus – A Metamodel for Computation*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[34] Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[35] Amr Sabry and Philip Wadler. A Reflection on Call-by-Value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997.

Calling paradigms and the box calculus

José Espírito Santo[1], Luís Pinto[1], and Tarmo Uustalu[2]

[1] Centro de Matemática, Universidade do Minho, Portugal
[2] Dept. of Computer Science, Reykjavik University, Iceland and Dept. of Software Science, Tallinn University of Technology, Estonia

In our previous work [2], we studied the computational meaning of the traditional maps of intuitionistic logic into S4 named after Girard and Gödel, respectively [6]. As source calculi we took, respectively, the ordinary (call-by-name, cbn) λ-calculus [1] and Plotkin's call-by-value (cbv) λ-calculus [5]; as target calculus we took a very simple extension of the λ-calculus with a S4 modality, where: the simplest solution to the problem of closure under substitution is adopted; the normalization steps relative to the modality are regarded as administrative and thus the syntax is organized so that such steps are integrated "on the fly" in the normalization steps for implication. The resulting calculus is what we name here *box calculus* (λ_{b} in [2]), and briefly recall now.

Terms of the box calculus are given by

$$M, N ::= \varepsilon(x) \mid \lambda x.M \mid MN \mid \mathsf{box}(N)$$

A term of the last form we call a *box*. The single reduction rule of the system is its β-rule:

$$(\lambda x.M)\mathsf{box}(N) \to [N/\varepsilon(x)]M \qquad (\beta_{\mathsf{b}})$$

The connection with modal logic is best seen in the typed version of the calculus. Types are given by

$$A ::= X \mid B \supset A \mid B \qquad\qquad B ::= \Box A$$

(note the restriction of antecedents of implications to *boxed types*), typing judgments have the form $x_1 : B_1, ..., x_n : B_n \vdash N : A$, and typing rules include:

$$\frac{}{\Gamma, x : \Box A \vdash \varepsilon(x) : A} \qquad\qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{box}(M) : \Box A}$$

A box is a term representing a proof ending with an inference introducing the box modality. A box is also the mandatory form for an argument to be passed to a function in the β-rule of the calculus. Finally, a box also marks a point where evaluation of proof terms seen as programs does not enter. The evaluation relation is given by closure of the base β_{b}-rule under applicative contexts only: reduction can happen both in function and argument positions of applications, but not under abstraction or inside a box – so evaluation is both weak and external. We say the calculus obeys the call-by-box (cbb) paradigm, which we see as unifying cbn and cbv: as shown in [2], the system enjoys a standardization theorem, from which we extract, via the modal embeddings, the standardization theorems for the two source calculi mentioned above.

In this abstract we report on two developments about the box calculus. The first concerns instantiations, an idea briefly introduced in [2]. These are translations of the box calculus giving a concrete implementation for the box modality in terms of the interpreting calculus. For instance, one can map the box calculus to the linear λ-calculus [4], implementing the box modality as the bang modality. We worked out a new instantiation into call-by-push-value [3], implementing the box modality as $F \circ U$, where F and U are the type operators of call-by-push-value shifting between computation and value types. This instantiation allows us to see in what

measure the subsumption of cbn and cbv offered by call-by-push-value is already contained in the unification offered by call-by-box.

As the second development reported in this abstract, we dig deeper in the unification offered by cbb. The treatment of cbn in [2] is so neat that we may say Girard's embedding just points out an isomorphic copy of the cbn λ-calculus as a fragment of the box calculus. We can now report an equally neat treatment for Plotkin's cbv λ-calculus. This requires refining slightly the box calculus, building in the untyped syntax a minimum of typing information—namely distinguishing between terms that can and cannot have a modal type. This creates in the box calculus two co-existing modes, the "left-first" and the "right-first", with which we can qualify the application constructor and reduction. The distinction between modes turns out to be connected to the distinction between calling paradigms. We verify that Girard's (resp. Gödel's) embedding can be recast as a map based on the idea of choosing the appropriate mode, translating application and reduction to left-first (resp. right-first) application and reduction. When thus recast, the modal embeddings deliver isomorphisms between the cbn (resp. Plotkin's cbv) λ-calculus and some neat fragment of the refined box calculus. In this sense, ordinary and Plotkin's λ-calculi truly co-exist inside a simple modal calculus.

Acknowledgments: J.E.S. and L.P. were partially financed by Portuguese Funds through FCT (Fundação para a Ciência e a Tecnologia) within the Projects UIDB/00013/2020 and UIDP/00013/2020 T.U. was supported by the Estonian Ministry of Education and Research institutional research grant IUT33-13. All three authors also benefited from the EU COST action CA15123 EUTYPES.

References

[1] H.P. Barendregt. *The Lambda Calculus*. North-Holland, 1984.

[2] J. Espírito Santo, L. Pinto, and T. Uustalu. Modal embeddings and calling paradigms. In H. Geuvers, editor, *4th Int. Conf. on Formal Structures for Computation and Deduction, FSCD 2019*, volume 131 of *Leibniz Int. Proc. in Informatics*, pages 18:1–18:20. Dagstuhl Publishing, 2019.

[3] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.

[4] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theor. Comput. Sci.*, 228(1-2):175–210, 1999.

[5] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.*, 1:125–159, 1975.

[6] A. Troelstra and H. Schwitchtenberg. *Basic Proof Theory*. Cambridge Univ. Press, 2000.

Finitary general type theories in computational form*

Andrej Bauer and Philipp G. Haselwarter

University of Ljubljana, Slovenia

A type theory can be presented in *declarative* or *algorithmic* style. The former specifies derivability of judgements by listing the inference rules, while the latter imbues the theory with a procedure for computing the derivable judgements. We present a computational calculus which, given a type theory in declarative style, computes the derivable judgements of that theory. The operational semantics of the calculus is inspired by bi-directional checking procedures which operate in two modes, one for synthesizing judgements and another for checking terms against types. The calculus is sound and complete for a class of type theories which we call the *finitary general type theories*, which are a special case of general type theories [2]. We have implemented an extended version of the calculus in the Andromeda 2 prover [1, 3].

Finitary general type theories The formalism of finitary general type theories encompasses a class of type theories in the style of Martin-Löf. They have judgement forms for asserting well-formedness of types and terms, and for equalities of types and terms, all hypothetical with respect to an intuitionistic typing context. Further conditions are imposed on the inference rules to guarantee reasonable meta-theoretic properties, such as uniqueness of typing, admissibility of substitution and derivability of presuppositions.

The derivation calculus presented below computes with judgements and boundaries, a notion which we explain briefly by example. Consider the usual formation rule for products, where we write $\{x\}\mathsf{B}(x)$ to indicate that x is bound in $\mathsf{B}(x)$:

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad x{:}\mathsf{A} \vdash \mathsf{B}(x)\ \mathsf{type}}{\vdash \Pi(\mathsf{A}, \{x\}\mathsf{B}(x))\ \mathsf{type}} \tag{1}$$

The object premises may be understood as providing "typing information" for the meta-variables, i.e., the first premise tells us that A is a type meta-variable, and the second that B is a type meta-variable depending on an argument of type A. We can display such typing information explicitly as follows:

$$\Pi \quad : \quad \frac{\mathsf{A} \ : \ \Box\ \mathsf{type} \qquad \mathsf{B} \ : \ \{x{:}\mathsf{A}\}\ \Box\ \mathsf{type}}{\Box\ \mathsf{type}} \tag{2}$$

The expressions "$\Box\ \mathsf{type}$" and "$\{x{:}\mathsf{A}\}\ \Box\ \mathsf{type}$" are *boundaries*, that is to say, judgements with a hole $\Box$, possibly appearing under a binder, as in the case of B above. In general, the hole $\Box$ of a boundary $\mathcal{B}$ may be filled with an expression e, called the *head*, to give a judgement $\mathcal{B}[e]$. Also note that it is sufficient to specify only the boundary of the conclusion in (2), as the missing head can be reconstructed as the symbol Π applied to the heads of the premises, suitably abstracted.

Each rule specifies a family of *closure rules*, which are obtained by instantiation of the meta-variables appearing in the rule with term and type expressions in context Γ. For example, by replacing A and B with type expressions $\Gamma \vdash A$ and $\Gamma, x{:}A \vdash B$ (note the change of font!) we obtain a closure rule

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, x{:}A \vdash B\ \mathsf{type}}{\Gamma \vdash \Pi(A, \{x\}B)\ \mathsf{type}} \tag{3}$$

*This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

Derivation $\mathcal{D}$	$::= x$	variable
	$\mid \mathsf{R}\,\mathcal{D}_1 \ldots \mathcal{D}_n$	application of rule R
	$\mid \{x{:}\mathcal{D}_1\}\,\mathcal{D}_2$	typed abstraction
	$\mid \{x\}\mathcal{D}$	untyped abstraction
	$\mid \mathcal{D}_1\{\mathcal{D}_2\}$	substitution
	$\mid \mathcal{D}_1\{\mathcal{D}_2\}_{\equiv}$	equality substitution

Figure 1: The core derivation calculus

One specifies a type theory by writing down declarative inference rules in the style of (1), while derivations are built inductively using closure rules in the style of (3).

This formalism encompasses many well-known type theories such as Martin-Löf type theory with products, sums, identity types, natural numbers, and a universe. One may add to that propositional truncation and univalence, or go in the other direction by postulating equality reflection. Type theories with non-standard judgement forms or special treatment of contexts, such as modal and cubical type theories, are not immediately expressible in our formalism, although one can often represent them faithfully by using universes.

The rules of a general type theory are declarative. They constitute a set of closure rules that specifies derivability of judgements, without suggesting any strategy for finding derivations. If we want to put the rules of a theory to work, for instance in a proof assistant, we need to formulate a corresponding computational version of the theory.

The derivation calculus We presume given a finitary general type theory $\mathcal{T}$. We present a *derivation calculus* whose operational semantics has two relations (whose rules are not shown in this abstract), similar to the synthesis and checking modes of bi-directional type checking:

$\mathcal{D} \Rrightarrow_\Gamma \mathcal{J}$	$\mathcal{D}$ synthesizes $\Gamma \vdash \mathcal{J}$
$\mathcal{D} \mathbin{@} \mathcal{B} \Rrightarrow_\Gamma e$	$\mathcal{D}$ checks $\mathcal{B}$ to derive $\Gamma \vdash \mathcal{B}[e]$

In the first case, an expression of the core derivation calculus $\mathcal{D}$ synthesizes a judgement $\Gamma \vdash \mathcal{J}$. In the second case $\mathcal{D}$ checks against a boundary $\mathcal{B}$ to produce a head e yielding a judgement $\Gamma \vdash \mathcal{B}[e]$. Thus $\mathcal{D}$, Γ and $\mathcal{B}$ are inputs, whereas $\mathcal{J}$ and e are outputs.

The syntax of the core calculus is shown in Figure 1. A derivation $\mathcal{D}$ may be a variable x from the given context Γ, an application of a rule R of the theory $\mathcal{T}$ to premises computed by the sub-derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$, a typed or untyped abstraction, or a substitution. The substitution expressions correspond to the two kinds of substitution rules in general type theories.

The operational semantics of derivations is well-behaved in the following sense:

Theorem. *The derivation calculus is sound and complete for any finitary general type theory $\mathcal{T}$: a judgement $\Gamma \vdash \mathcal{J}$ is derivable in $\mathcal{T}$ if, and only if, there is $\mathcal{D}$ such that $\mathcal{D} \Rrightarrow_\Gamma \mathcal{J}$.*

The core derivation calculus is amenable to extensions and adaptations that make it more practically useful. For example, when the type theory $\mathcal{T}$ has decidable equality checking we may elide derivations that check judgmental equalities, as these can be replaced by calls to an equality-checking algorithm. In Andromeda 2 the derivation calculus is embedded in a more elaborate meta-level programming language with computational effects that supports user-defined techniques of proof development.

References

[1] The Andromeda proof assistant. http://www.andromeda-prover.org.

[2] Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. Toward an initiality theorem for general type theories. In *Workshop on Types, Homotopy Type Theory, and Verification*. Hausdorff Institute of Mathematics – Bonn (Germany), June 2018.

[3] Andrej Bauer, Philipp G. Haselwarter, and Anja Petković. A generic proof assistant. In *Foundations and Applications of Univalent Mathematics – Herrsching (Germany)*, December 2019.

10 Types, logic and lambda calculi

Comparing Session Type Interpretations of Linear Logic*

Bas van den Heuvel and Jorge A. Pérez

University of Groningen, Groningen, The Netherlands
{b.van.den.heuvel,j.a.perez}@rug.nl

Context *Session types* are a popular approach to typed message-passing concurrency [14, 15, 20]. A session type describes the types and order of messages exchanged along a channel. For example, the session type !int.?bool.end types a channel meant to perform the following sequence: send an integer, receive a boolean, and close the channel. Session types have been widely studied for the π-calculus [18,19], the paradigmatic model of concurrency and interaction.

Girard developed linear logic as the next logical step for reasoning about computation [11]. At an early stage of its development, he noted that linear logic would be a good candidate for a Curry-Howard correspondence for concurrency [13]. Following attempts by Abramsky [1] and others, Caires and Pfenning found a correspondence between a session-typed π-calculus and *intuitionistic* linear logic [7]. Our example session type !int.?bool.end can be written as the proposition $\mathsf{int} \otimes (\mathsf{bool} \multimap \mathbf{1})$, where $\otimes$ is sending, $\multimap$ is reception, and $\mathbf{1}$ is closing. Shortly after, Wadler found a correspondence using *classical* linear logic [22], in which our example can be written similarly to the intuitionistic proposition, but with $\parr$ as reception: $\mathsf{int} \otimes (\mathsf{bool} \parr \mathbf{1})$.

The logical interpretations of session types discovered in [7] and [22] have been extended to provide justifications for notions such as behavioural polymorphism [5], (co)recursion [16, 21], cyclic connections [10], non-determinism [4], domain-awareness [6], conflation of types [3], and more. These extensions actually form a *family* of logical interpretations of session types, each based on either intuitionistic or classical linear logic. Based on these developments, Atkey [2] observed: "[j]ust as the Iron Curtain [...] lead to the same work being done twice, once in the East and once in the West, the existence of two logically-based session-typed concurrency formalisms [...] means that analogous work is performed on both sides."

This Work In ongoing work, we aim at formally comparing the session type systems derived from Curry-Howard interpretations of classical linear logic [9, 22] and intuitionistic linear logic [8], respectively referred to as CLL and ILL. To this end, we have developed United Linear Logic (ULL), a logic based on the linear fragment of the Logic of Unity (LU) [12], developed by Girard to study classical, intuitionistic, and linear logic together in one system. Following a similar spirit, we have designed ULL to subsume both CLL and ILL. Indeed, ULL defines a basic framework of reference in which both type systems can be objectively compared.

We briefly discuss differences between LU and ULL. LU uses polarities in propositions to distinguish between classical and intuitionistic fragments. Since polarities are not relevant for the linear part of LU, they are not included in ULL. Also, in ULL we do not include rules in LU that allow linear propositions to freely switch sides in sequents. Moreover, UL does not fully support propositions $\bot$ and $\mathbf{1}$, which are required to interpret the terminated session type end. For this reason, ULL includes complementary rules for these propositions. Finally, ULL includes an additional cut rule and identity axiom, which are necessary to fully subsume CLL.

ULL consists of two-sided sequents $\Gamma; \Delta \vdash \Lambda$, where Δ and Λ are linear contexts (propositions that must be used exactly once) and Γ is the unrestricted context. The key insight of the Curry-Howard correspondences in [7, 22] is that these sequents can be annotated with π-calculus

*Work partially supported by the Netherlands Organization for Scientific Research (NWO) under the VIDI Project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

process terms and channel names (the free names of the process term). In ULL, the cut rule then looks as follows:

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, x : A \qquad \Gamma; \Delta', x : A \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta' \vdash (\nu x)(P \mid Q) :: \Lambda, \Lambda'} \text{(CutR)}$$

Here, P and Q both have a free channel x. In both sequents, x has type A. However, it appears on opposite sides of the turnstile, so P and Q implement complementary behaviors on x. Therefore, we can place them in parallel ($P \mid Q$) and bind their common channel ($(\nu x)(\dots)$).

Our design for ULL is *symmetrical*, in the sense that everything that can be done on one side of the turnstile can be done on the other. As a result, in ULL there are two cut rules and two identity axioms. For example, the above cut rule works on the right of the turnstile and is standard in intuitionistic linear logic, whereas the following (symmetrical) rule works on the left, using *duality* (the notion that behaviour on opposite ends of a channel should be opposite) indicated by $(\cdot)^\perp$, as is standard in classical linear logic:

$$\frac{\Gamma; \Delta, x : A \vdash P :: \Lambda \qquad \Gamma; \Delta', x : A^\perp \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta' \vdash (\nu x)(P \mid Q) :: \Lambda, \Lambda'} \text{(CutL)}$$

ULL can type all processes typable in CLL and CLL can type all processes typable in ULL. This means that, in essence, ULL is a two-sided presentation of CLL.

The two-sidedness of ULL is very convenient for a formal comparison with ILL. Sequents in ILL are very similar to those of ULL, but they require the linear context on the right to consist of *exactly one proposition*: $\Gamma; \Delta \vdash A$. If we place the same restriction on typing derivations in ULL, certain rules become impossible to use. Consider, for example, above CutL rule. If both assumptions have one channel-type pair on the right ($|\Lambda| = |\Lambda'| = 1$), then the consequent would have two channel-type pairs on the right ($|\Lambda, \Lambda'| = |\Lambda| + |\Lambda'| = 2$), which clearly violates our restriction. It turns out that the rules that remain usable coincide exactly with the rules of ILL, showing that all ILL-typable processes are typable in ULL (and thus in CLL).

The fact that ULL has more inference rules than ILL is evidence that not all ULL-typable processes are typable in ILL. Consider the following process:

$$P = x(y).!y(z).P',$$

where $x(y)$ is an input and $!y(z)$ is a *replicated* input. P can be typed in ULL in several ways, but all of them use rules that violate the restriction of having exactly one channel-type pair on the right. Hence, P is not typable in ILL, and so CLL is (slightly) more expressive than ILL.

Locality Caires, Pfenning, and Toninho were the first to observe a difference in expressivity between CLL and ILL [9]. As they explain, ILL enforces *locality*, a principle well-known in the process calculi literature. As defined by Merro [17], "[t]he locality property [...] is achieved by imposing that only the output capability of [channels] may be transmitted, i.e., the recipient of a [channel] may only use it in output actions." As illustration, consider again process P above: it receives a channel y which is then used to deploy a replicated server. Clients connect to such a server by sending a channel to it, so the received service y has to receive a channel, thus violating the locality principle. Caires, Pfenning, and Toninho use P as a specific example to explain that ILL enforces the locality principle for shared names, while CLL does not. Using ULL as reference framework, we can now use P as a counter-example to *prove* the difference in expressivity between CLL and ILL, thus formalizing the observation by Caires, Pfenning, and Toninho [9].

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1):3–57, April 1993.

[2] Robert Atkey. Observed Communication Semantics for Classical Processes. In Hongseok Yang, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 56–82, Berlin, Heidelberg, 2017. Springer.

[3] Robert Atkey, Sam Lindley, and J. Garrett Morris. Conflation Confers Concurrency. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, pages 32–55. Springer International Publishing, Cham, 2016.

[4] Luis Caires and Jorge A. Pérez. Linearity, Control Effects, and Behavioral Types. In *Programming Languages and Systems*, pages 229–259. Springer, 2017.

[5] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 330–349, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Domain-Aware Session Types. In Wan Fokkink and Rob van Glabbeek, editors, *30th International Conference on Concurrency Theory (CONCUR 2019)*, volume 140 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, pages 222–236. Springer Berlin Heidelberg, 2010.

[8] Luís Caires, Frank Pfenning, and Bernardo Toninho. Towards concurrent type theory. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 1–12. ACM, January 2012.

[9] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, March 2016.

[10] Ornela Dardha and Simon J. Gay. A New Linear Logic for Deadlock-Free Session-Typed Processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 91–109. Springer International Publishing, 2018.

[11] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987.

[12] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3):201–217, February 1993.

[13] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT '87*, Lecture Notes in Computer Science, pages 52–66, Berlin, Heidelberg, 1987. Springer.

[14] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[15] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[16] Sam Lindley and J. Garrett Morris. Talking Bananas: Structural Recursion for Session Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 434–447, Nara, Japan, 2016. ACM.

[17] Massimo Merro. Locality and Polyadicity in Asynchronous Name-Passing Calculi. In Jerzy Tiuryn, editor, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 238–251, Berlin, Heidelberg, 2000. Springer.

[18] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[19] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, October 2003.

[20] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[21] Bernardo Toninho, Luis Caires, and Frank Pfenning. Corecursion and Non-divergence in Session-Typed Processes. In Matteo Maffei and Emilio Tuosto, editors, *Trustworthy Global Computing*, Lecture Notes in Computer Science, pages 159–175, Berlin, Heidelberg, 2014. Springer.

[22] Philip Wadler. Propositions As Sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286, Copenhagen, Denmark, 2012. ACM.

On self-interpreters for the $\lambda^{\Box}$-calculus and other modal λ-calculi

Miëtek Bak

IMDEA Software Institute
mietek.bak@imdea.org

It is commonly believed that a total programming language cannot have a self-interpreter. Following Brown and Palsberg's [5] discovery of a self-interpreter for System F_{ω}, Bauer [3] explains that whether such a language can or cannot have a self-interpreter depends on the notion of self-interpreter. I refine the notions supplied by Bauer and reveal that the strongly normalizing $\lambda^{\Box}$-calculus of Davies and Pfenning [6] and other modal λ-calculi already have a self-interpreter. The result sheds new light on an old problem in meta-programming.

Definition 1. An *impredicative self-interpreter* is given by a type $\circledast$ of *source code of programs of arbitrary type*, and for all types τ, a meta-level *quoting function* $\ulcorner\text{-}\urcorner_{\tau}$ from programs of type τ to programs of type $\circledast$, and a *universal program* $\mathtt{u}_{\tau} : \circledast \supset \tau$ such that $\mathtt{u}_{\tau}\ulcorner e \urcorner \equiv_{\beta} e$ for all programs $e : \tau$.

Read *program* as a closed expression of a λ-calculus. Bauer's "typed self-interpreter" is equivalent to the definition of an impredicative self-interpreter.

Theorem 2. *If a λ-calculus has an impredicative self-interpreter, then it has fixed-point operators at all types.*

The above theorem justifies the belief that a total programming language cannot have a self-interpreter. Bauer [4] supplies a constructive proof. Similar results are shown by Brown and Palsberg, Harper [7], Hoare and Allison [8], and McBride [9].

Normal modal logics include the necessity connective $\Box$, the necessitation rule $\frac{\vdash \tau}{\vdash \Box \tau}$, and the distribution axiom $\Box(\sigma \supset \tau) \supset \Box\sigma \supset \Box\tau$. Modal logics such as T, B, S4, and S5 are normal, and include the reflexivity axiom $\Box\tau \supset \tau$. The $\lambda^{\Box}$-calculus corresponds to intuitionistic S4, and extends the simply typed λ-calculus with the terms $\mathsf{box}\, e$ and $\mathsf{let\,box}\, v = e_1 \,\mathsf{in}\, e_2$, and the following typing and computation rules:

$$\frac{v : \tau \in \Delta}{\Delta; \Gamma \vdash v : \tau}\,\mathrm{mvar} \qquad \frac{\Delta; \cdot \vdash e : \tau}{\Delta; \Gamma \vdash \mathsf{box}\, e : \Box\tau}\,\Box\mathrm{I} \qquad \frac{\Delta; \Gamma \vdash e_1 : \Box\tau \quad \Delta, v : \tau; \Gamma \vdash e_2 : \sigma}{\Delta; \Gamma \vdash \mathsf{let\,box}\, v = e_1 \,\mathsf{in}\, e_2 : \sigma}\,\Box\mathrm{E}$$

$$\frac{}{\mathsf{let\,box}\, v = \mathsf{box}\, e_1 \,\mathsf{in}\, e_2 \mapsto e_2[v/e_1]}\,\Box\beta$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{let\,box}\, v = e_1 \,\mathsf{in}\, e_2 \mapsto \mathsf{let\,box}\, v = e_1' \,\mathsf{in}\, e_2}\,\mathrm{letbox}_1 \qquad \frac{e_2 \mapsto e_2'}{\mathsf{let\,box}\, v = e_1 \,\mathsf{in}\, e_2 \mapsto \mathsf{let\,box}\, v = e_1 \,\mathsf{in}\, e_2'}\,\mathrm{letbox}_2$$

The program $\mathsf{box}\, e : \Box\tau$ is β-normal for all programs $e : \tau$, as no congruence rule applies.

Definition 3. A *predicative self-interpreter* is given by, for all types τ, a type $\Box\tau$ of *source code of programs of type τ* such that $\Box\tau \neq \tau$, a meta-level *quoting function* $\ulcorner\text{-}\urcorner_{\tau}$ from programs of type τ to programs of type $\Box\tau$, and a *universal program* $\mathtt{u}_{\tau} : \Box\tau \supset \tau$ such that $\mathtt{u}_{\tau}\ulcorner e \urcorner_{\tau} \equiv_{\beta} e$ for all programs $e : \tau$.

In an impredicative self-interpreter, there is a single source code type, but in a predicative self-interpreter, there is a ramified hierarchy of source code types. Therefore, unlike Bauer's "weak (Brown-Palsberg) self-interpreter", the definition of a predicative self-interpreter already rejects the trivial quoting function $\ulcorner e \urcorner_{\tau} = e$ without additional constraints.

Definition 4. The quoting function in a predicative self-interpreter is

1. *normal*: when, for all types τ, the program $\ulcorner e \urcorner_\tau : \Box\tau$ is β-normal for all programs $e : \tau$;
2. *distributive*: when, for all types σ and τ, there is a program $\mathtt{d}_{\sigma,\tau} : \Box(\sigma \supset \tau) \supset \Box\sigma \supset \Box\tau$ such that $\mathtt{d}_{\sigma,\tau}\ \ulcorner e_1 \urcorner_{\sigma,\tau}\ \ulcorner e_2 \urcorner_\sigma \equiv_\beta \ulcorner e_1\ e_2 \urcorner_\tau$ for all programs $e_1 : \sigma \supset \tau$ and $e_2 : \sigma$;
3. *weakly acceptable*: when, for all types τ, there is a meta-level Gödel-encoding function $\#_\tau$ from programs of type $\Box\tau$ to programs of type $\mathtt{Nat}$ such that $\#_\tau \ulcorner e \urcorner_\tau \equiv_\beta \underline{\mathtt{g}(e)}$ for all programs $e : \tau$;
4. *strongly acceptable*: when, for all types τ, there is a Gödel-encoding program $\mathtt{g}_\tau : \Box\tau \supset \mathtt{Nat}$ such that $\mathtt{g}_\tau \ulcorner e \urcorner_\tau \equiv_\beta \underline{\mathtt{g}(e)}$ for all programs $e : \tau$.

Read $\mathtt{g}(e)$ as a Gödel encoding of the program e by a natural, and $\underline{n}$ as a program of type $\mathtt{Nat}$ that represents the natural n. Weak and strong acceptability are two possible readings of Bauer's "acceptability". Bauer does not discuss weak acceptability. Bauer's "strong (Brown-Palsberg) self-interpreter" is equivalent to a predicative self-interpreter with a normal and strongly acceptable quoting function.

Normality requires the quoting function to produce values. Both weak and strong acceptability require the quoting function to preserve the unevaluated source code of the quoted program for analysis. Weak acceptability only requires allowing analysis by means external to the language, but strong acceptability also requires allowing analysis by internal means.

Theorem 5. *The $\lambda^{\Box}$-calculus has a predicative self-interpreter with a normal and distributive quoting function.*

Proof. Define $\ulcorner e \urcorner_\tau = \mathsf{box}\, e$, $\mathtt{u}_\tau = \lambda x : \Box\tau.\ \mathsf{let\,box}\, v = x\, \mathsf{in}\, v$, and $\mathtt{d}_{\sigma,\tau} = \lambda x : \Box(\sigma \supset \tau).\ \lambda y : \Box\sigma.\ \mathsf{let\,box}\, v = x\, \mathsf{in}\ \ \mathsf{let\,box}\, w = y\, \mathsf{in}\ \ \mathsf{box}\ (v\ w)$. Normality and distributivity follow from the definition of computation rules. □

A similar proof applies to any *box calculus*: a λ-calculus corresponding to a normal modal logic with reflexivity, such as the simply typed fragment of contextual modal type theory given by Nanevski, Pfenning, and Pientka [10], and the intensional λ-calculus of Artemov and Bonelli [1].

Theorem 6. *The $\lambda^{\Box}$-calculus, extended with naturals, has a predicative self-interpreter with a normal, distributive, and weakly acceptable quoting function.*

Proof. Proceed as above. Weak acceptability is shown by induction on typing derivations. □

A similar proof applies to any *intensional box calculus*: a box calculus, extended with naturals, in which computation does not proceed under box.

Any remaining controversy about the question whether a total programming language can or cannot have a self-interpreter will amount to disagreement about the "correct" meaning of the unqualified phrase "self-interpreter". Instead, observe that by refining the notion of self-interpreter, we recover a specification of one of Sheard's [11] research challenges in meta-programming. A language that has a predicative self-interpreter with a strongly acceptable quoting function is also a language that supports *intensional analysis of code*. Bauer shows that System T is such a language.

Whether System T is a practical meta-programming language is immaterial. Knowing intensional analysis of code can be supported in a total programming language, we can now search for such a language that will also be practical for the purposes of meta-programming. The family of intensional box calculi is a promising place to start.

I conjecture there is a strongly normalizing intensional box calculus that supports intensional analysis of code, and is sound and complete with respect to a Kripke semantics similar to the one given for the $\lambda^{\Box}$-calculus in my previous work [2].

Acknowledgments I am grateful to Guillaume Allais, Ahmad Salim Al-Sibahi, Andrej Bauer, Jacques Carette, Dan Doel, Paolo G. Giarrusso, Tom Jack, Conor McBride, and Aleksandar Nanevski, for helpful comments and discussion.

References

[1] Sergei N. Artemov and Eduardo Bonelli. The intensional lambda calculus. In Sergei N. Artemov and Anil Nerode, editors, *International Symposium on Logical Foundations of Computer Science (LFCS 2007), Proceedings*, volume 4514 of *Lecture Notes in Computer Science*, pages 12–25. Springer, 2007.

[2] Miëtek Bak. Introspective Kripke models and normalisation by evaluation for the $\lambda^{\Box}$-calculus, 2017. *7th Workshop on Intuitionistic Modal Logic and Applications (IMLA 2017)*. Available at http://github.com/mietek/imla2017.

[3] Andrej Bauer. On self-interpreters for Gödel's System T. In Silvia Ghelizan and Jelena Ivetić, editors, *22nd International Conference on Types for Proofs and Programs (TYPES 2016), Book of Abstracts*, pages 23–24, 2016.

[4] Andrej Bauer. On self-interpreters for System T and other typed λ-calculi, 2016. Available at http://math.andrej.com/wp-content/uploads/2016/01/self-interpreter-for-T.pdf.

[5] Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for F_{ω}. In Rastislav Bodík and Rupak Majumdar, editors, *43rd Symposium on Principles of Programming Languages (POPL 2016), Proceedings*, pages 5–17. ACM, 2016.

[6] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *23rd Symposium on Principles of Programming Languages (POPL 1996), Conference Record*, pages 258–270. ACM, 1996.

[7] Robert Harper. *Practical Foundations for Programming Languages*. CUP, 2016.

[8] C. A. R. Hoare and Donald C. S. Allison. Incomputability. *ACM Computing Surveys*, 4(3):169–178, 1972.

[9] Conor McBride. On termination, 2003. Available at http://mail.haskell.org/pipermail/haskell-cafe/2003-May/004343.html.

[10] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

[11] Tim Sheard. Accomplishments and research challenges in meta-programming. In *2nd International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001), Proceedings*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer, 2001.

Multimodal Dependent Type Theory

Daniel Gratzer[1], G. A. Kavvos[1], Andreas Nuyts[2], and Lars Birkedal[1]

[1] Aarhus University, Denmark
[2] imec-DistriNet, KU Leuven, Belgium

Abstract

We introduce MTT, a dependent type theory which supports multiple modalities. MTT is parametrized by a mode theory which specifies a collection of modes, modalities, and transformations between them. We show that different choices of mode theory allow us to use the same type theory to compute and reason in many modal situations, including guarded recursion, axiomatic cohesion, and parametric quantification.

In order to increase the expressivity of Martin-Löf Type Theory we often wish to extend it with new connectives, and in particular with unary type operators that we call *modalities* or *modal operators*. The addition of a modality to a dependent type theory is a non-trivial exercise: modal operators often interact with the context of a type or term in a complicated way, and naïve approaches lead to undesirable interplay with other type formers and substitution. This work is concerned with the development of a systematic approach to the formulation of type theories that feature multiple modalities.

Rather than designing a new dependent type theory for some preordained set of modalities, we will introduce a system that is *parametrized* by a *mode theory*, i.e. an algebraic specification of a modal situation. This system, which we call MTT, solves two problems. First, like Licata et al. [3, 4], we address the lack of generality of state-of-the-art solutions in modal type theory. By instantiating MTT with different mode theories, we show that it can capture a wide class of situations. Some of these, e.g. the one for guarded recursion, lead to a previously unknown system that improves upon earlier work. Second, the predictable behavior of our rules allows us to prove metatheoretic results about large classes of instantiations of MTT at once. For example, our main metatheoretic result—viz. canonicity—applies irrespective of the chosen mode theory.

MTT is not just multimodal, but also *multimode* [3]. That is, each judgment of MTT can be construed as existing in a particular *mode*. All modes have some things in common—e.g. there will be dependent sums in each—but some might possess distinguishing features. From a semantic point of view, different modes correspond to different context categories. In this light, modalities intuitively correspond to *functors* between those categories: in fact, they will be structures slightly weaker than *dependent right adjoints* (DRAs) [2].

At a high level, MTT can be thought of as a machine that converts a concrete description of modes and modalities into a type theory. This description, which is often called a *mode theory*, is given in the form of a *small strict 2-category* [6, 4] using the following correspondence:

$$\begin{aligned} \text{object} &\sim \text{mode} \\ \text{morphism} &\sim \text{modality} \\ \text{2-cell} &\sim \text{natural map between modalities} \end{aligned}$$

In order to describe the syntax of MTT, we fix a strict 2-category $\mathcal{M}$. At each mode $m \in \mathcal{M}$ we have a standard Martin-Löf Type Theory. For example, the judgment $\Gamma\ \mathsf{ctx}\ @\ m$ states that Γ is a well-formed context in mode m.

The modal structure of MTT follows a *Fitch-style* discipline: each morphism $\mu : m \to n$ of $\mathcal{M}$ induces an operation on contexts in the *reverse* direction. We will denote this by a *lock*:

$$\textsc{cx/lock}\ \frac{\Gamma\ \mathsf{ctx} \mathbin{@} m}{\Gamma, \mathbin{🔒}_\mu\ \mathsf{ctx} \mathbin{@} n}$$

These lock operations come with equations that make them contravariantly functorial in the modality $\mu : m \to n$. Each modality also induces a modal operator $\langle \mu \mid - \rangle$ on types. Intuitively, $\mathbin{🔒}_\mu$ behaves like a left adjoint to $\langle \mu \mid - \rangle$. Just as with DRAs [2], the MTT formation and introduction rules for modal types effectively *transpose* types and terms across this adjunction:

$$\textsc{tp/modal}\ \frac{\Gamma, \mathbin{🔒}_\mu \vdash A\ \mathsf{type} \mathbin{@} n}{\Gamma \vdash \langle \mu \mid A \rangle\ \mathsf{type} \mathbin{@} m} \qquad \textsc{tm/modal-intro}\ \frac{\Gamma, \mathbin{🔒}_\mu \vdash M : A \mathbin{@} n}{\Gamma \vdash \mathsf{mod}_\mu(M) : \langle \mu \mid A \rangle \mathbin{@} m}$$

In order to obtain a well-behaved syntax, every variable in the context will be annotated with a modality, i.e. will be of the form $x : (\mu \mid A)$. Intuitively a variable $x : (\mu \mid A)$ is the same as a variable $x : \langle \mu \mid A \rangle$, but the annotations—signalled by rounded parentheses—are part of the structure of a context. The variable rule allows us to use $x : (\mu \mid A)$ if the locks to its right are 'strong enough,' and that strength is witnessed by a 2-cell of $\mathcal{M}$:

$$\textsc{tm/var}\ \frac{\mu : m \to n \qquad \alpha : \mu \Rightarrow \mathsf{locks}(\Gamma_1)}{\Gamma_0, x : (\mu \mid A), \Gamma_1 \vdash x^\alpha : A^\alpha \mathbin{@} m}$$

Here, $\mathsf{locks}(\Gamma_1)$ composes all the modalities of the locks in Γ_1, and $(-)^\alpha$ for $\alpha : \mu \Rightarrow \nu$ is an admissible operation such that $\Gamma, \mathbin{🔒}_\mu \vdash A\ \mathsf{type} \mathbin{@} m$ implies $\Gamma, \mathbin{🔒}_\nu \vdash A^\alpha\ \mathsf{type} \mathbin{@} m$. Finally, the elimination rule, which is reminiscent of the dual context style [5], navigates the difference between $x : \langle \mu \mid A \rangle$ and $x : (\mu \mid A)$ by essentially implementing a kind of *modal induction*:

$$\textsc{tm/modal-elim}\ \frac{\begin{array}{cc}\nu : m \to o & \mu : n \to m \qquad \Gamma, x : (\nu \mid \langle \mu \mid A \rangle) \vdash B\ \mathsf{type}_1 \mathbin{@} o \\ \Gamma, \mathbin{🔒}_\nu \vdash M_0 : \langle \mu \mid A \rangle \mathbin{@} m & \qquad \Gamma, x : (\nu \circ \mu \mid A) \vdash M_1 : B[\mathsf{mod}_\mu(x)/x] \mathbin{@} o\end{array}}{\Gamma \vdash \mathsf{let}_\nu\ \mathsf{mod}_\mu(x) \leftarrow M_0\ \mathsf{in}\ M_1 : B[M_0/x] \mathbin{@} o}$$

In addition to introducing the syntax of MTT, we also formulated its semantics in the style of Awodey's *natural models* [1]. We showed that MTT may be used to reason about several important modal settings, and proven basic metatheorems about its syntax including *canonicity*, which we show via categorical gluing. In the future we plan to further develop the metatheory of MTT. We specifically hope to prove that MTT enjoys normalization, and hence that type-checking is decidable—provided that the mode theory is. This result would pave the way to a practical implementation of a multimodal proof assistant.

Acknowledgments

Alex Kavvos was supported in part by a research grant (12386, Guarded Homotopy Type Theory) from the VILLUM Foundation. Andreas Nuyts holds a PhD Fellowship from the Research Foundation - Flanders (FWO). This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

References

[1] Steve Awodey. Natural models of homotopy type theory. 28(2):241–286, 2018.

[2] Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. 2018. To appear in Mathematical Structures in Computer Science.

[3] Daniel R. Licata and Michael Shulman. *Adjoint Logic with a 2-Category of Modes*, pages 219–235. Springer International Publishing, 2016.

[4] Daniel R. Licata, Michael Shulman, and Mitchell Riley. A Fibrational Framework for Substructural and Modal Logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 2017.

[5] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

[6] Jason Reed. A Judgmental Deconstruction of Modal Logic. 2009. Manuscript.

A distributed term assignment for dual-intuitionistic logic

Gianluigi Bellin[1] and Luca Tranchini[2]

[1] University of Verona, gianluigi.bellin@univr.it
[2] University of Tübingen, luca.tranchini@gmail.com

Bi-intuitionistic logic [6] is a conservative extension of intuitionistic logic with a binary operator of subtraction $A - B$ (informally read as "A but not B"). Although in (Kripke-)semantics subtraction is easily definable as the dual of intuitionistic implication, the proof theory of bi-intuitionistic logic turns out to be a very delicate matter: as of today no plain sequent-calculus enjoying cut-elimination is known, although several well-behaving "enriched" sequent calculi has been recently developed (for nested, labelled and deep inference calculi see [2, 4, 5]).

In [3], Crolard proposes a term-assignment for bi-intuitionistic logic by first defining in the classical $\lambda\mu$-calculus $A - B$ as $A \wedge \neg B$, and then imposing some restrictions on the typing rules to "constructivize" the logic. According to Crolard, such restrictions capture a notion of "safety" for the co-routines encoding the proofs of $A - B$. However, the details of Crolard's presentation are rather intricate, and the notion of safe co-routine is not very intuitive.

In the present paper, we propose a term-assignment for the subtractive fragment of Crolard's system (i.e. in wich subtraction is the only operator) using primitive operations with three distinctive features. First, our term calculus has a distributed nature: whereas in the $\lambda\mu$-calculus all conclusions but one are μ-variables, in our calculus the computational content is distributed among the conclusions, i.e. to each conclusion one assigns a possibly complex term. Second, reduction is also distributed, in the sense that the reduction acts globally on a set of terms, and not on a single one. Third, typing judgements have the form $x : C \blacktriangleright \Delta$, i.e. a set of terms is typed by declaring a single variable (somewhat dualizing the simply typed λ-calculus, where a single term is typed declaring a set of variables). A version of the calculus for a linear variant of subtraction was presented in [1].

Given an infinite set of variables $x, y, z, \ldots$, terms, list of terms and error messages are defined as follows:

Terms	t, s, r	$::=$	$x \mid \mathtt{mkc}(t, x) \mid x_{\langle t \rangle}$
Lists of terms	l	$::=$	$[] \mid l \cdot t$
Error messages	e, f	$::=$	$\mathtt{err}(t \mid x \mapsto l)$

The free occurrences of x in l are bound in $\mathtt{err}(t \mid x \mapsto l)$, and the displayed occurrences of x (though not the free occurrences of x possibly occurring in t) are bound in $\mathtt{mkc}(t, x)$ and $x_{\langle t \rangle}$.

Clearly the set of free variables of each term contains exactly one free variable. As it will be clear from the typing rules below, the same is true of error messages occurring in a typing derivation: the only error messages $\mathtt{err}(t \mid x \mapsto l)$ we will consider will be those in which all terms in l depend on x, and hence, according to the stipulation on the free variables just given, the only free variable in an error message $\mathtt{err}(t \mid x \mapsto l)$ will be the free variable of t.

The result of substituting a term s for a variable x in a term t (resp. list of terms l, error message e), notation $t[\![s/x]\!]$ (resp. $l[\![s/y]\!]$, $e[\![s/y]\!]$) is defined in the obvious way.

Typing judgements have the form $x : C \blacktriangleright \Delta$, where the $x : C$ is called the *antecedent*, and Δ the *succedent*. Δ is a set of typed terms and error messages $\{t_1 : A_1, \ldots, t_n : A_n, e_1, \ldots, e_n\}$.

The typing rules are the following, where if $l = t_1 \cdot \ldots \cdot t_n$ we write $l : B, \Delta$ for $t_1 : B, \ldots, t_n : B, \Delta$ (or simply for Δ if $n = 0$):

$$\frac{}{x : A \blacktriangleright x : A}\ \text{Axiom}$$

$$\frac{x : C \blacktriangleright t : A, \Delta}{x : C \blacktriangleright \mathtt{mkc}(t, y) : A - B, y_{\langle t \rangle} : B, \Delta}\ -\text{I} \qquad \frac{y : A \blacktriangleright l : B, \Delta}{x : A - B \blacktriangleright \mathtt{err}(x \mid y \mapsto l), \Delta[\![y_{\langle x \rangle}/y]\!]}\ -\text{E}$$

If there is a derivation of $x : C \blacktriangleright \Delta$ we say that Δ is a *family* of typed terms under the variable declaration $x : C$. The different members of a family can be thought of encoding alternative hypothetical, or defeasible, scenarios that can be computed from the variable declaration. In the introduction rule, the premise of the rule says that given a proof of C we could compute beside certain alternative hypothetical scenarios Δ, a further one in which we constructed a proof t of A. The rule tells us that this scenario can be split up into two alternative scenarios, one in which a proof of B has been constructed (possibly using the proof t of A), and one in which no proof of B can be obtained (this is the scenario represented by the formula $A - B$ in the succedent of the conclusion). In the elimination rule, the premise tells us that given a proof y of A we could construct certain alternative hypothetical scenarios Δ and further ones in each of which we have obtained a proof of B. From this, given a proof x of $A - B$ we could construct variants of the scenarios Δ (since when one has a proof of $A - B$ one has one of A as well), however, the scenarios in which we had obtained proofs of B are "incompatible" with our having a proof of $A - B$. These are therefore "suppressed" in the succedent of the conclusion of the rule and replaced by an error message registering the incompatibility between the threads of computation $y \mapsto l$ leading from the proof of A to those of B, and the proof x of $A - B$.

Redexes are error message of the form $\mathtt{err}(\mathtt{mkc}(t, x) \,|\, y \mapsto l)$. In contrast with what happens in the λ-calculus, reduction does not simply rewrite terms on terms, but family of typed terms onto family of typed terms and it can be expressed as follows:

$$\frac{x : C \ \blacktriangleright \ t : B \,,\, \Delta \qquad z : B \blacktriangleright l : A \,,\, \Delta'}{x : C \ \blacktriangleright \ \mathtt{err}(\mathtt{mkc}(t, y) \mid z \mapsto l) \,,\, \Delta \,,\, \Delta'[{}^{z\langle \mathtt{mkc}(t,y)\rangle}/_{z}] \,,\, y_{\langle t\rangle} : A \ \rightsquigarrow \ \Delta \,,\, \Delta'[{}^{t}/_{z}] \,,\, l[t/z] : A}\ \beta$$

A full axiomatization of β-reduction is obtained by adding further rules expressing the fact that β-reduction is a congruence. Subject reduction holds in the following form:

Lemma 0.1. *If $x : C \blacktriangleright \Delta \rightsquigarrow \Delta^*$, then each term in Δ^* has the same type as one in Δ.*

We prove strong normalization through an embedding into the simply typed λ-calculus.

In particular, we define an embedding $()^\circ$ that leaves atomic types untouched and that maps $A - B$ onto $B^\circ \supset A^\circ$ and such that for every family of typed terms $\Delta \equiv e_1, \ldots, e_m, t_1 : A_1, \ldots, t_n : A_n$ under the variable declaration $x : C$ and for every variable declaration $\Gamma \equiv z_1 : A_1^\circ, \ldots, z_n : A_n^\circ$, Δ°_Γ is a λ-term of type C° under the variable declaration Γ.

Proposition 0.2. *If $x : C \blacktriangleright \Delta \rightsquigarrow \Delta^*$, where $\Delta \equiv e_1 \ldots e_n, t_1 : A_1, \ldots t_m : A_m$ then for all variable declarations $\Gamma \equiv y_1 : A_1^\circ, \ldots y_m : A_m^\circ$ we have that $y_1 : A_1^\circ, \ldots, y_m : A_m^\circ \rhd \Delta^\circ_\Gamma \rightsquigarrow \Delta^{*\circ}_\Gamma$.*

The embedding above consists in a sort of duality between the simply typed λ-calculus and our distributed term calculus, which in turn suggests an alternative way of using the latter, namely to encode an intuitionistic notion of refutability rather than a dual-intuitionistic notion of provability. Graphically, the idea can be exemplified by using our terms to decorate Prawitz's rules of intuitionistic natural deduction from the bottom to the top, thereby expressing a backwards reading of intuitionistic rules: given refutations of the conclusions one computes refutations of the premises:

$$\frac{\mathtt{mkc}(t, x) : A \supset B \qquad x_{\langle t\rangle} : A}{t : B} \qquad \frac{\boxed{\mathtt{err}(t \mid x \mapsto s_i)}}{\begin{array}{c}[s_i^* : A] \\ x_{\langle t\rangle} : B \\ \hline t : A \supset B\end{array}}$$

where the error message $\mathtt{err}(t \mid x \mapsto s_i)$ plays the role of a discharge index in linking the assumptions with the inference rule at which they are discharged.

References

[1] Gianluigi Bellin. Categorical proof theory of co-intuitionistic linear logic. *Logical Methods in Computer Science*, 10(3), 2014.

[2] Linda Buisman and Rajeev Goré. A cut-free sequent calculus for bi-inutitionistic logic. In N. Olivetti, editor, *TABLEAUX 2007*. Springer, 2007.

[3] Tristan Crolard. A formulae-as-types interpretation of subtractive logic. *Journal of Logic and Computation*, 14(4):529–570, 2004.

[4] Luís Pinto and Tarmo Uustalu. Proof search and counter-model construction for bi-intuitionistic propositional logic with labelled sequents. In Martin Giese and Arild Waaler, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 5607 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 2009.

[5] Linda Postniece. Deep inference in bi-intuitionistic logic. In H. Ono, M. Kanazawa, and de Queiroz R., editors, *Logic, Language, Information and Computation. WoLLIC 2009*. Springer.

[6] Cecylia Rauszer. A formalization of the propositional calculus of H-B logic. *Studia Logica*, 33(1):23–34, 1974.

11 Logic, category and types

An Induction Principle for Cycles

Nicolai Kraus and Jakob von Raumer

Consider a graph. Imagine we want to prove a certain property for every path within the graph, where a path is a sequence $(v_0, e_0, v_1, e_1, \dots, v_k)$ of edges e_i from the vertex v_i to the vertex v_{i+1}. The obvious first approach is induction: We show that the property holds for the empty path, and that adding an edge at the end of a path preserves the property. We now make the situation more challenging by changing it slightly:

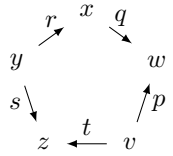
Problem 1. Imagine we want to prove a certain property for every *cycle*, i.e. every *closed* path within the graph (a path satisfying the condition $v_0 = v_k$). How can this be approached?

An example for a property could be the statement that every vertex in a cycle has even degree, something which is not true for paths in general. Straightforward induction does not work in the situation of Problem 1 anymore, since a cycle can in general not be created from a smaller cycle by adding an edge (cycles are simply not inductively generated).

This situation occurs in the context of coherence conditions in homotopy type theory.[1] Given a *set* (a 0-truncated type) A and a binary proof-relevant relation on A, i.e. a type family $(\rightsquigarrow) : A \to A \to \mathsf{Type}$, recall that the property of the *set-quotient* $A/\!\rightsquigarrow$ is that a function $g : A/\!\rightsquigarrow \to B$ is uniquely given by a function $f : A \to B$ which respects $\rightsquigarrow$, i.e. comes together with $h : \Pi\{x, y : A\}.(x \rightsquigarrow y) \to f(x) = f(y)$. This property is only guaranteed as long as B is a *set*. Sometimes, this condition not satisfied. We show in the paper [3] the following result:

Lemma 2. *If B is a 1-type and A is a set, then a function $g : A/\!\rightsquigarrow \to B$ is uniquely given by a triple (f, h, c) with f, h as above, and c witnessing that any* closed zig-zag *(or simply* cycle*) of $\rightsquigarrow$ is sent by h to a trivial equality* refl *in B.*

The coherence condition c says, for example, that if we are given $v, w, x, y, z : A$ with $p : v \rightsquigarrow w$, $q : x \rightsquigarrow w$, $r : y \rightsquigarrow x$, $s : y \rightsquigarrow z$, $t : v \rightsquigarrow z$ as in the diagram on the right, then the equality $h(p) \cdot h(q)^{-1} \cdot h(r)^{-1} \cdot h(s) \cdot h(t)^{-1} : f(v) = f(v)$ is equal to $\mathsf{refl}_{f(v)}$.



An example for a set-quotient in homotopy type theory is the explicit construction of the *free group* as in [6, Thm 6.11.17]. For a given set Y, we set $A :\equiv \mathsf{List}(Y + Y)$. We think of the left copy of Y as positive and the right as negative, and write $\_^{-1} : Y + Y \to Y + Y$ for the "swap" operation. The relation of interest is then generated by

$$[x_0, \dots, x_{i-1}, x_i, x_i^{-1}, x_{i+1}, \dots, x_n] \rightsquigarrow [x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_n], \tag{1}$$

and $\mathsf{List}(Y + Y)/\!\rightsquigarrow$ has the correct universal property of the free group on Y. It is an open question (a variation of the long-standing unsolved problem recorded in [6, Ex 8.2]) whether $\mathsf{List}(Y + Y)/\!\rightsquigarrow$ is equivalent to the *free higher group*, namely the loop space of the wedge of Y-many circles; details can be found in [3]. Constructing functions from the set-quotient $\mathsf{List}(Y+Y)/\!\rightsquigarrow$ into a 1-type is the key to showing an approximation to the mentioned question. To do this, we would like to apply Lemma 2. However, on its own, this lemma is not useful

[1] The situation described in the first paragraph is of course reminiscent of a central observation of (homotopy) type theory, namely the fact that the equality eliminator J (a.k.a. *path induction*) does not imply Streicher's K (which one could call *loop induction*): with J, we can only replace an equality by refl if it is a general equality between two different points. This is not the connection that we study in this work.

precisely because of Problem 1: the condition that any cycle is sent to a trivial equality is hard to verify. Overcoming this difficulty is the motivation for this talk. The title of the current talk description alludes to an approach to Problem 1 that we present in [3].

If we look at (1) (and the relevant relations in similar examples), we can observe that the relation $\rightsquigarrow$ has two properties (one of them proof-relevant) that are familiar from the theory of reduction systems in general (for example, see [1]). First, (1) is *co-wellfounded* (a.k.a. *Noetherian*): this is clear, since we can only reduce a finite number of times until no redexes of the form $[x, x^{-1}]$ are left. The general definition is:

Definition 3 (accessibility [6, Chp. 10.3])**.** The family $\mathsf{acc}^{\rightsquigarrow} : A \to \mathsf{Type}$ is generated inductively by a single constructor $\mathsf{step} : \Pi(a : A).(\Pi(x : A).(x \rightsquigarrow a) \to \mathsf{acc}^{\rightsquigarrow}(x)) \to \mathsf{acc}^{\rightsquigarrow}(a)$. We say that a is *accessible* if we have $\mathsf{acc}^{\rightsquigarrow}(a)$, and if all a are accessible, then $\rightsquigarrow$ is *wellfounded*. We say that $\rightsquigarrow$ is *Noetherian* if $\rightsquigarrow^{\mathsf{op}}$ is wellfounded.

Second, the relation (1) is *locally confluent*: If we have two redexes and reduce one, we can still reduce the other (or both reductions arrive at the same result). In general:

Definition 4 (local confluence)**.** We say that $\rightsquigarrow$ is *locally confluent* if, for any span, there is a matching extended cospan. This means that, given $x, y, z : A$ with $x \rightsquigarrow y$ and $x \rightsquigarrow z$, we have $w : A$ such that $y \rightsquigarrow^* w$ and $z \rightsquigarrow^* w$.

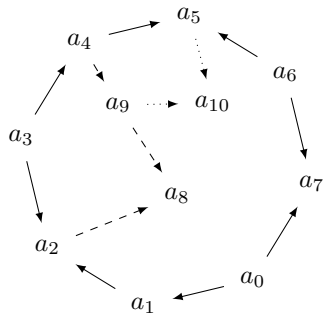
The key construction in our paper [3] is the following. Starting from a relation $\rightsquigarrow$ on A, we construct a new relation $\rightsquigarrow^\circ$ on the type of cycles of $\rightsquigarrow$. This new relation is given by a variation of the *multiset extension*, which is known to preserve wellfoundedness (a proof essentially building on Definition 3 has been given by Nipkow [5]). In a nutshell, for cycles α and β, we have $\alpha \rightsquigarrow^\circ \beta$ if α can be transformed into β by removing one vertex (one element $a_0 : A$) and replacing it by a finite number $b_1, \ldots, b_k$ of vertices such that we have $a_0 \rightsquigarrow b_i$ for all i.

Lemma 5. *If $\rightsquigarrow$ is Noetherian on A, then so is $\rightsquigarrow^\circ$ on the type of cycles. If $\rightsquigarrow$ is in addition locally confluent, then any cycle α can be split (see illustration below) into a cycle β with $\alpha \rightsquigarrow^\circ \beta$, and a cycle which is given by a single span and local confluence.*

Let now a type family Q indexed over cycles be given (e.g. the type witnessing that any cycle gets mapped to the trivial equality, as in Lemma 2). Under some natural assumption (Q respects "merging" and "rotating" of cycles), the above lemmas give us the following principle:

Theorem 6 (Noetherian cycle induction [3])**.** *Assume $\rightsquigarrow$ is Noetherian and locally confluent. Assume further that Q is inhabited at every empty cycle and at every cycle that comes from a span and the local confluence property. Then, Q is inhabited at every cycle.*

On the right is an illustration of *Noetherian cycle induction*. Assume we want to show a property Q for the big octagon $a_0 - a_7$. We first "remove" the confluence cycle spanned by $a_2 \leftsquigarrow a_3 \rightsquigarrow a_4$ to get the nonagon consisting of $a_0 - a_9$ without a_3 but with the dashed edges: we now have more vertices, but the nonagon is smaller than the octagon in the order $\rightsquigarrow^\circ$. In the next step, we "remove" the confluence cycle spanned by $a_9 \leftsquigarrow a_4 \rightsquigarrow a_5$, and so on, until the empty cycle is reached.



This induction principle allows us to approach questions involving set-quotients such as the problem of the free higher group mentioned above; in particular, we can re-prove the result of [2]. We also believe that there are use-cases outside of type theory, e.g. for graph rewriting as in [4].

References

[1] Gérard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27:797–821, 1980.

[2] Nicolai Kraus and Thorsten Altenkirch. Free higher groups in homotopy type theory. In *Symposium on Logic in Computer Science (LICS 2018)*, pages 599–608. ACM, 2018.

[3] Nicolai Kraus and Jakob von Raumer. Coherence via wellfoundedness: Taming set-quotients in homotopy type theory. *ArXiv e-prints*, 2020.

[4] Michael Löwe. Van-Kampen pushouts for sets and graphs. Technical report, 2010. Fachhochschule für die Wirtschaft Hannover.

[5] Tobias Nipkow. An inductive proof of the wellfoundedness of the multiset order. Unpublished note, 1998.

[6] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book/, 2013.

Metatheoretic proofs internally to presheaf categories*

Rafaël Bocquet[1], Ambrus Kaposi[1], and Christian Sattler[2]

Eötvös Loránd University, Budapest, Hungary
bocquet@inf.elte.hu and akaposi@inf.elte.hu
University of Nottingham, United Kingdom
sattler.christian@gmail.com

Introduction Proofs of the metatheoretic properties of dependent type theories and other typed logics and languages, such as proofs of canonicity, normalization, gluing, parametricity or various translations between theories, involve complicated inductions over the syntax of the theory. We present a way to structure such proofs by working in the internal type-theoretic languages of suitable presheaf categories.

Internal models of type theory We use categories with families (CwFs) [4, 2] equipped with additional structure as the models of our type theories. They consist of a category $\mathcal{C}$, equipped with presheaves of types and terms, objects representing the empty context and context extensions, along with natural transformations for each type-theoretic operation and equations between them. Most of the additional structure on $\mathcal{C}$ can concisely be described in the type-theoretic internal language of the presheaf category $\widehat{\mathcal{C}}$. This observation is used in some of the existing general definitions of type theories [1, 9]. For example, the presheaves of types and terms, and the $\mathbb{N}$ and Π type formers can be specified in the internal language of $\widehat{\mathcal{C}}$ as follows:

$$\begin{array}{lcl} \mathsf{Ty} & : & \mathsf{Set} \\ \mathsf{Tm} & : & \mathsf{Ty} \to \mathsf{Set} \\ \mathbb{N} & : & \mathsf{Ty} \\ \Pi & : & (A : \mathsf{Ty}) \to (B : \mathsf{Tm}\ A \to \mathsf{Ty}) \to \mathsf{Ty} \end{array}$$

The type $(\mathsf{Tm}\ A \to \mathsf{Ty})$ of the argument B of Π is a presheaf function type: using the internal language of presheaf categories is a way to interpret higher-order abstract syntax (HOAS). Π-types are usually given externally by a map $\Pi_\Gamma : (A : \mathsf{Ty}_\Gamma) \to \mathsf{Ty}_{\Gamma \rhd A} \to \mathsf{Ty}_\Gamma$, natural in Γ, but the properties of the context extension operation $(- \rhd -)$ imply that the internal and external definitions are equivalent.

Only the empty context and the context extension operations can not directly be described internally, unless we use the interpretation of *crisp type theory* [8, 6] in $\widehat{\mathcal{C}}$ and its comonadic modality $\flat$.

Internal dependent models The generalized algebraic presentation of CwFs automatically provides an initial model $\mathcal{S}$ satisfying an induction principle: there is a dependent section from $\mathcal{S}$ to any dependent model over $\mathcal{S}$. The definition of dependent model can be derived mechanically from the QIIT-signature presenting the type theory [5]. By applying a similar transformation

*The first author was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). The second author was supported by the ÚNKP-19-4 New National Excellence Program of the Ministry for Innovation and Technology and by the Bolyai Fellowship of the Hungarian Academy of Sciences. The third author was supported by USAF grant FA9550-16-1-0029.

to the internal definition of models, we define a notion of dependent model internal to $\widehat{\mathcal{S}}$:

$$\begin{array}{lcl}
\mathsf{Ty}^\bullet & : & \mathsf{Ty} \to \mathsf{Set} \\
\mathsf{Tm}^\bullet & : & \{A\}(A^\bullet : \mathsf{Ty}^\bullet\ A)(a : \mathsf{Tm}\ A) \to \mathsf{Set} \\
\mathbb{N}^\bullet & : & \mathsf{Ty}^\bullet\ \mathbb{N} \\
\Pi^\bullet & : & \{A\}(A^\bullet : \mathsf{Ty}^\bullet\ A)\{B\}(B^\bullet : \{a\}(a^\bullet : \mathsf{Tm}^\bullet\ A^\bullet\ a) \to \mathsf{Ty}^\bullet\ (B\ a)) \to \mathsf{Ty}^\bullet\ (\Pi\ A\ B)
\end{array}$$

Internal and external dependent models do not exactly correspond to each other, but we can still reconstruct an external dependent model from any internal one, and then obtain, externally, a dependent section of the reconstructed external model.

A proof of canonicity based on logical predicates can be given as an internal dependent model in the internal language of $\widehat{\mathcal{S}}$:

$$\begin{array}{lcl}
\mathsf{Ty}^\bullet\ A & :\equiv & \mathsf{Tm}\ A \to \mathsf{Set} \\
\mathsf{Tm}^\bullet\ A^\bullet & :\equiv & \lambda(a : \mathsf{Tm}\ A) \mapsto A^\bullet\ a \\
\mathbb{N}^\bullet & :\equiv & \lambda(n : \mathsf{Tm}\ \mathbb{N}) \mapsto (m : \mathbf{N}) \times (n = \mathsf{suc}^m\ \mathsf{zero}) \\
\Pi^\bullet\ A^\bullet\ B^\bullet & :\equiv & \lambda(f : \mathsf{Tm}\ (\Pi\ A\ B)) \mapsto (a : \mathsf{Tm}\ A)(a^\bullet : A^\bullet\ a) \to B^\bullet\ (\mathsf{app}\ f\ a)
\end{array}$$

Internal induction principles The presheaf category $\widehat{\mathcal{S}}$ is not a nice setting for more complicated proofs, such as normalization proofs: it forces all of our constructions to be stable under all substitutions, but normal forms are only stable under renamings. To fix this, we change the base category. For normalization, we work in the presheaf category $\widehat{\mathcal{G}}$ over the comma category $\mathcal{G} = (\mathcal{S} \downarrow F)$, where $F : \mathcal{R} \to \mathcal{S}$ is the CwF morphism from the CwF of renamings $\mathcal{R}$ to $\mathcal{S}$.

The presheaf category $\widehat{\mathcal{G}}$ has many good properties: the CwF structures of $\mathcal{R}$ and $\mathcal{S}$ can faithfully be transported over $\mathcal{G}$, and the CwF morphism $F : \mathcal{R} \to \mathcal{S}$ can also faithfully be encoded. Furthermore, we can distinguish in $\widehat{\mathcal{G}}$ the presheaves that come from $\widehat{\mathcal{R}}$ or $\widehat{\mathcal{S}}$; and call them $\mathcal{R}$-discrete or $\mathcal{S}$-discrete presheaves. Type-theoretically, they are accessible reflective subuniverses of the universe of all presheaves. Moreover, the $\mathcal{R}$-/$\mathcal{S}$-discrete presheaves can be identified with the discrete types arising from interpretations of spatial type theory [8] in $\widehat{\mathcal{G}}$. In particular, we have an adjoint pair of modalities $(\Box \dashv \Diamond)$, where the comonadic modality $\Box$ classifies the $\mathcal{R}$-discrete presheaves. This means that we can reuse the theory of modalities developed in [7, 8] in this setting.

We can then use the $\Box$ modality to define *F-relative internal dependent models*, which encode inductions over the syntax whose results are only stable under renamings. We can define and prove an induction principle for F-relative dependent models.

A recent normalization proof by Coquand [3] can be translated to this framework by defining a suitable F-relative dependent model. We can prove normalization and the decidability of equality for types and terms fully internally, without ever working explicitly with contexts, substitutions or renamings in the proof.

Agda formalization[1] We have formalized in Agda internal proofs of canonicity and normalization for a reasonably large dependent type theory (including Π-types with the η rule, booleans, natural numbers, identity types and a universe closed under the other type formers).

References

[1] Paolo Capriotti. *Models of type theory with strict equality.* PhD thesis, University of Nottingham, UK, 2017.

[1] https://gitlab.com/RafaelBocquet/internal_metatheory/tree/master/Agda

[2] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019.

[3] Thierry Coquand. Canonicity and normalization for dependent type theory. *Theor. Comput. Sci.*, 777:184–191, 2019.

[4] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995.

[5] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *PACMPL*, 3(POPL):2:1–2:24, 2019.

[6] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal universes in models of homotopy type theory. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[7] Egbert Rijke, Michael Shulman, and Bas Spitters. Modalities in homotopy type theory. *CoRR*, abs/1706.07526, 2017.

[8] Michael Shulman. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Mathematical Structures in Computer Science*, 28(6):856–941, 2018.

[9] Taichi Uemura. A general framework for the semantics of type theory. *CoRR*, abs/1904.04097, 2019.

Elementary doctrines as coalgebras

Jacopo Emmenegger[1], Fabio Pasquali[2], and Giuseppe Rosolini[3]

DIMA, Università di Genova, Italy

[1]jacopo.emmenegger@edu.unige.it [2]pasquali@dima.unige.it [3]rosolini@unige.it

Abstract

We show that the inclusion of the 2-category of elementary doctrines into that of primary doctrines is 2-comonadic. Coalgebras are doctrines of descent data on equivalence relations. Several examples are mentioned.

Lawvere's hyperdoctrines mark the beginning of applications of category theory in logic, and they provide a very clear algebraic tool to work with syntactic theories and their extensions in logic. Recall that a ***primary doctrine*** is a functor $P\colon \mathcal{C}^{\mathrm{op}} \longrightarrow \mathbf{Pos}$ such that (i) the base category $\mathcal{C}$ has finite products, (ii) for every object A of $\mathcal{C}$, the poset $P(A)$ has finite meets and, (iii) for every arrow $f : A \longrightarrow B$, the monotone function $f^* = P(f) : P(B) \longrightarrow P(A)$, called reindexing along f, preserves finite meets.

A logical theory T in a multi-sorted language with conjunctions gives rise to a primary doctrine P_T as follows. The base category consists contexts and context morphisms, *i.e.* finite lists of typed variables and finite lists of typed terms. Composition is given by substitution of terms in terms and product is concatenation of contexts. The poset $\mathsf{P}_T(x_1 : S_1, \ldots, x_n : S_n)$ consists of formulas in the context $(x_1 : S_1, \ldots, x_n : S_n)$, where

$$\phi \leq \psi \quad \text{iff} \quad x_1 : S_1, \ldots, x_n : S_n \,|\, \phi \vdash_T \psi.$$

Meets are given by conjunctions and reindexing along a list of terms is given by substitution of terms in formulas. Conversely, any primary doctrine gives rise to a theory in a multi-sorted language with conjunctions.

A primary doctrine is ***elementary*** when it has left adjoints to reindexing along parametrised diagonals $I \times A \longrightarrow I \times A \times A$ satisfying suitable Beck-Chevalley and Frobenius stability conditions, see [8]. A more concrete description is provided by the following characterisation, which we could find no reference for. A primary doctrine $P\colon \mathcal{C}^{\mathrm{op}} \longrightarrow \mathbf{ISL}$ is elementary if and only if for each object A in $\mathcal{C}$, there is an object $\mathfrak{d}_A$ in $P(A \times A)$ such that

(a) $\mathrm{pr}_1^*(\alpha) \wedge \mathfrak{d}_A \leq \mathrm{pr}_2^*(\alpha)$ for every A and every α in $P(A)$;

(b) $\top_A \leq \langle \mathrm{id}_A, \mathrm{id}_A \rangle^*(\mathfrak{d}_A)$ for every A;

(c) $\langle \mathrm{pr}_1, \mathrm{pr}_3 \rangle^*(\mathfrak{d}_A) \wedge \langle \mathrm{pr}_2, \mathrm{pr}_4 \rangle^*(\mathfrak{d}_B) \leq \mathfrak{d}_{A \times B}$ for every A and B.

When the language of a theory T has equality, the associated doctrine P_T is elementary and the object $\mathfrak{d}_A$ is the (equivalence class) of the equality predicate. Conversely, the theory associated to an elementary doctrine has equality.

A morphism of primary doctrines $P \longrightarrow P'$ is a product-preserving functor F between the base categories together with a meet-preserving natural transformation $P \dashrightarrow P'F$. A morphism of doctrines is the same thing as an interpretation between the associated theories.

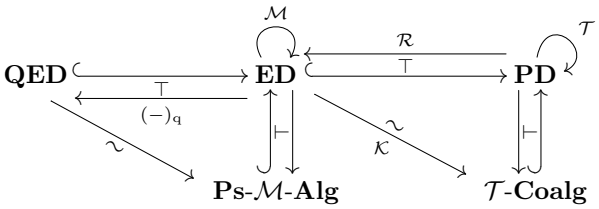
It is possible to arrange primary doctrines in a 2-category **PD**, which is equivalent to the 2-category of faithful fibrations with fibred products, see [4, 16]. The equivalence restricts between the 2-category of elementary doctrines **ED** and the 2-category of faithful fibrations with equality. A morphism in **PD** between elementary doctrines is a morphism in **ED**, *i.e.* the associated interpretation of theories preserves equality, if and only if it preserves the object $\mathfrak{d}_A$ for every A.

In [7] an extension of a logical theory to allow quotient types is described in terms of a completion $(-)_{\mathrm{q}}\colon \mathbf{ED} \longrightarrow \mathbf{QED}$, where $\mathbf{QED}$ is the full 2-subcategory of $\mathbf{ED}$ on elementary doctrines with quotients. In [17] the completion is proved to be pseudo-monadic, that is, quotients are (pseudo) algebraic structure on elementary doctrines. In [12] the second author analysed that construction and showed that in fact it provided a right adjoint $\mathcal{R}\colon \mathbf{PD} \longrightarrow \mathbf{ED}$ to the 2-full inclusion of elementary doctrines into primary doctrines. The base category of the doctrine $\mathcal{R}(P)$ consists of equivalence relations $\rho \in P(A \times A)$ and extensional functions, and the fibres are subposets on descent data, *i.e.* those $\alpha \in P(A)$ such that $\mathrm{pr}_1^*\alpha \wedge \rho \leq \mathrm{pr}_2^*\alpha$.

Write $\mathcal{T}\colon \mathbf{PD} \longrightarrow \mathbf{PD}$ for the 2-comonad induced by the adjunction and $\mathcal{T}$-**Coalg** for the 2-category of coalgebras. We extend the analysis in [12] and prove the following.

Theorem. *The canonical comparison 2-functor* $\mathcal{K}\colon \mathbf{ED} \longrightarrow \mathcal{T}$-**Coalg** *is an isomorphism.*

The embedding $\mathbf{ED} \hookrightarrow \mathbf{PD}$ is then 2-comonadic. It follows that the monad on $\mathcal{T}$-**Coalg** associated to the right-hand vertical adjunction below is isomorphic, via $\mathcal{K}$, to the monad $\mathcal{M}$ on $\mathbf{ED}$ generated by $(-)_{\mathrm{q}}$.



The functor $\mathcal{R}$ occurs in a number of different situations, as suggested by the following examples.

1. Several constructions of categories of (partial) equivalence relations can be factored through the functor $\mathcal{R}$, in combination with other completions of doctrines, see [8, 9, 10]. Major examples are the exact completion of a regular category and of a category with (weak) finite limits [1], the tripos-to-topos construction [3] and various forms of setoid model of dependent type theories such as Martin-Löf type theory (MLTT) [11], the Calculus of Constructions [2] and the Minimalist Foundation [6]. The last two examples are instances of quotient completions that do not produce exact categories.

2. A first order theory T eliminates imaginaries in the sense of Poizat [13] if, for every formula $x, y : A \,|\, \rho(x, y)$ which is provably in T an equivalence relation, there is a formula $x : A,\ z : B \,|\, \phi(x, z)$ such that
$$T \vdash (\forall x : A)(\exists! z : B)(\forall y : A)(\rho(x, y) \leftrightarrow \phi(x, z)).$$
Given a first order theory T, the (first order) theory $\overline{T}$ associated to the doctrine $\mathcal{R}(\mathsf{P}_T)$ eliminates imaginaries. Moreover, every model of the original theory T can be turned functorially into a model of $\overline{T}$ and, when T has equality, this extension is conservative. In particular, it enjoys the same properties of Shelah's T^{eq} construction, see [15], which may be regarded as the concatenation of $\mathcal{R}$ with other constructions.

3. Other examples come from generalisations of metric spaces as in [5] and [14]. For an inf-semilattice H consider the primary doctrine $\mathsf{H}^{(-)}\colon \mathcal{S}et^{\mathrm{op}} \longrightarrow \mathbf{ISL}$ whose fibres are posets of H-valued functions. For H the positive real line $[0, \infty)$ with the opposite order, the base category of $\mathcal{R}(\mathsf{H}^{(-)})$ is the category of pseudo ultrametric spaces. Taking H to be the closed unit interval $[0, 1]$ with the opposite of order, the base category of $\mathcal{R}(\mathsf{H}^{(-)})$ is the category of 1-bounded pseudo ultrametric spaces.

References

[1] A. Carboni and E.M. Vitale. Regular and exact completions. *J. Pure Appl. Algebra*, 125:79–117, 1998.

[2] T. Coquand and G. Huet. The calculus of constructions. *Inform. and Comput.*, 76:95–120, 1988.

[3] J.M.E. Hyland, P.T. Johnstone, and A.M. Pitts. Tripos Theory. *Math. Proc. Camb. Phil. Soc.*, 88:205–232, 1980.

[4] B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the foundations of mathematics*. North Holland Publishing Company, 1999.

[5] F.W. Lawvere. Metric spaces, generalized logic, and closed categories. *Rend. Sem. Mat. Fis. Milano*, 43:135–166, 1973.

[6] M.E. Maietti. A minimalist two-level foundation for constructive mathematics. *Ann. Pure Appl. Logic*, 160(3):319–354, 2009.

[7] M.E. Maietti and G. Rosolini. Elementary quotient completion. *Theory Appl. Categ.*, 27:445–463, 2013.

[8] M.E. Maietti and G. Rosolini. Quotient completion for the foundation of constructive mathematics. *Log. Univers.*, 7(3):371–402, 2013.

[9] M.E. Maietti and G. Rosolini. Unifying exact completions. *Appl. Categ. Structures*, 23:43–52, 2015.

[10] M.E. Maietti and G. Rosolini. Relating quotient completions via categorical logic. In Dieter Probst and Peter Schuster (eds.), editors, *Concepts of Proof in Mathematics, Philosophy, and Computer Science*, pages 229–250. De Gruyter, 2016.

[11] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984. Notes by G. Sambin of a series of lectures given in Padua, June 1980.

[12] F. Pasquali. A co-free construction for elementary doctrines. *Appl. Categ. Structures*, 23(1):29–41, Feb 2015.

[13] B. Poizat. Une théorie de Galois imaginaire. *J. Symbolic Logic*, 48(4):1151–1170 (1984), 1983.

[14] J.J.M.M. Rutten. Elements of generalized ultrametric domain theory. *Theoret. Comput. Sci.*, 170(1):349 – 381, 1996.

[15] S. Shelah. *Classification theory and the number of nonisomorphic models*, volume 92 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, second edition, 1990.

[16] T. Streicher. Fibred categories. Available at `arXiv:1801.02927`, 2019.

[17] D. Trotta. Completions of elementary doctrines and pseudo-distributive laws. Manuscript, submitted, 2019.

Bi-Intuitionistic Types via Alternating Contexts

Ranald Clouston[1] and Ian Shillito[2]

[1] Research School of Computer Science, Australian National University, Canberra, Australia
ranald.clouston@anu.edu.au
[2] ian.shillito@anu.edu.au

Bi-Intuitionistic Logic (BIL), introduced in the 1970s by Rauszer [10], extends intuitionistic propositional logic with a binary operator $-\!\prec$, dual to $\rightarrow$, called *exclusion* (sometimes *subtraction*, or *co-implication*). BIL is a conservative extension of intuitionistic logic, but the disjunction property does not hold. This suggests that a proof-theoretic treatment may require multiple conclusions, as for classical logic.

BIL can be given semantics via the usual Kripke models for intuitionistic logic. A formula $\phi -\!\prec \psi$ is satisfied at a point if there exists a predecessor (i.e. a point related via the *converse* of the accessibility relation) that satisfies ϕ but falsifies ψ. In classical logic, $\phi -\!\prec \psi$ can be identified with $\phi \wedge \neg\psi$. The fact that the accessibility relation can be travelled along in both ways shows that BIL is similar to tense logics.

As with many tense logics, the proof theory of BIL has proved to be notoriously hard to treat. An early attempt at a standard sequent calculus by Rauszer [9] was claimed to enjoy cut-elimination. However a counterexample was discovered by Pinto and Uustalu [8]: the valid sequent $\phi \vdash \psi, \chi \rightarrow ((\phi -\!\prec \psi) \wedge \chi)$ is not derivable without the cut rule. This has led to the development of more exotic proof systems for BIL, such as display calculi [5], nested sequents [6], labelled sequents [8], and proof nets [12]. The only work that we are aware of that suggests a formulae-as-types interpretation for BIL is that of Crolard [3]. Unfortunately Crolard states that the cut-elimination of Rauszer's sequent calculus follows from his results; therefore his work must be erroneous, although we are yet to discover the precise source of error.

Crolard's computational interpretation for BIL was motivated by a desire to suggest control operators powerful enough to express **catch** and **throw** operators, but less powerful than the classical **call/cc**, which some consider harmful [7]. He uses de Groote's type system for classical logic [4] as a basis, with new rules for $-\!\prec$:

$$\frac{\Gamma \vdash \Delta, M : \phi \quad \Gamma, x : \chi \vdash \Delta, C[x] : \psi}{\Gamma \vdash \Delta, \alpha : \psi, \mathbf{make\text{-}coroutine}\, M\, C_\alpha : \phi -\!\prec \chi} \qquad \frac{\Gamma \vdash \Delta, M : \phi -\!\prec \psi \quad \Gamma, x : \phi \vdash \Delta, N : \psi}{\Gamma \vdash \Delta, \mathbf{resume}\, M\, \mathbf{with}\, x \mapsto N : \chi}$$

Crolard must further adjust other rules of de Groote's system to avoid capturing classical logic. In particular the lambda rule

$$\frac{\Gamma, x : \phi \vdash \Delta, M : \psi}{\Gamma \vdash \Delta, \lambda x.M : \phi \rightarrow \psi}$$

is obviously not intuitionistically valid. Restricting this rule to require Δ to be empty would be intuitionistically valid, but the calculus would no longer be closed under substitution, and hence, β-reduction. Crolard instead defines a syntactic notion of *safe* terms, based on restrictions on occurrences of variables in the scope of coroutines, and restricts the type system for BIL to include only those terms.

We can motivate our desire to move beyond Crolard's approach in the following ways. First, Crolard claims that his system provides a new proof of a result now known to be wrong, namely cut-elimination for Rauszer's sequent calculus. Second, Crolard's calculus contains a **let** rule for cuts which does not appear to be in general eliminable.

Third, the notion of safeness is very complex. Fourth, the safeness restriction is a global property, rather than local to each typing rule, which forces one to examine the entire term to verify each application of a rule. This makes it difficult to see how to conduct backward proof search or local type checking and inference.

Our approach

We are investigating a new approach to types for BIL that is simpler than that of Crolard, and in particular devoid of global restrictions on terms. We take inspiration from Fitch-style modal λ-calculi (see e.g. Clouston [1]), and modify the usual notion of a variable context. A *λ-variable context* Γ is as usual a comma-separated list of typed variables, where commas are understood as products. A *μ-variable context* Δ is likewise, except we understand commas as sums, and use Greek letters for the variable names. An *alternating context* Θ alternates between λ- and μ- contexts, i.e. we take the union of

$$\Theta_\lambda := \Gamma \mid \Theta_\mu, \Gamma \qquad\qquad \Theta_\mu := \Theta_\lambda \blacktriangleleft \Delta$$

Here comma is understood as a product, and $\blacktriangleleft$ is understood as *exclusion*. Alternating contexts with single conclusions are more expressive than standard contexts with multiple conclusions: in BIL, $\Gamma \vdash \Delta, A$ is logically equivalent to $\Gamma \blacktriangleleft \Delta \vdash A$, but types in alternating contexts with many layers of alternation are not in general equivalent to some type in context with multiple conclusions (although they are in classical logic). Our rules for exclusion then become:

$$\frac{\Theta \blacktriangleleft \alpha : \psi, \Delta \vdash M : \phi}{\Theta \blacktriangleleft \alpha : \psi, \Theta' \vdash \mathbf{make\text{-}coroutine}\, M\, \alpha : \phi \prec \psi} \qquad \frac{\Theta \vdash M : \phi \prec \psi \quad x : \phi \blacktriangleleft \Delta \vdash N : \psi}{\Theta; \Theta' \vdash \mathbf{resume}\, M\, \mathbf{with}\, x \mapsto N : \chi}$$

where ; is either comma or $\blacktriangleleft$, and in both rules Θ' is a weakening of Δ. By adapting the other rules of de Groote to both an intuitionistic setting and alternating contexts, one gets a sound and complete cut-free system for BIL. As a consequence, Pinto and Uustalu's counterexample is derivable in our system without the use of a cut rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \phi \blacktriangleleft \alpha : \psi \vdash x : \phi}}{x : \phi \blacktriangleleft \alpha : \psi, y : \chi \vdash \mathbf{make\text{-}coroutine}\, x\, \alpha : (\phi \prec \psi)} \quad \dfrac{}{x : \phi \blacktriangleleft \alpha : \psi, y : \chi \vdash y : \chi}}{x : \phi \blacktriangleleft \alpha : \psi, y : \chi \vdash \langle \mathbf{make\text{-}coroutine}\, x\, \alpha, y \rangle : (\phi \prec \psi) \wedge \chi}}{x : \phi \blacktriangleleft \alpha : \psi \vdash \lambda x. \langle \mathbf{make\text{-}coroutine}\, x\, \alpha, y \rangle : \chi \to ((\phi \prec \psi) \wedge \chi)}$$

This is work in progress, where the most important piece of future work is to define a reduction relation on terms, inspired by those of de Groote and Crolard, proving desirable results such as confluence, strong normalisation, and the subformula property. We also wish to investigate possible computational applications of the constructors for $\prec$, following the ideas sketched by Crolard. Finally, we wish to investigate categorical semantics. The naive semantics, where we take a category with both Cartesian closure and its dual, are degenerate [2, 11], so as with classical logic it is an interesting question how to proceed.

Acknowledgments. We are grateful to Luca Tranchini and Gianluigi Bellin for interesting discussions on this topic, and to the anynomous reviewers for their useful comments.

References

[1] Ranald Clouston. Fitch-style modal lambda calculi. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 258–275, Cham, 2018. Springer International Publishing.

[2] Tristan Crolard. Subtractive logic. *Theoretical Computer Science*, 254:1-2:151–185, 2001.

[3] Tristan Crolard. A formulae-as-types interpretation of Subtractive Logic. *Journal of Logic and Computation*, 14:4:529–570, 2004.

[4] Philippe de Groote. Strong normalization of classical natural deduction with disjunction. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, pages 182–196, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[5] Rajeev Goré. Dual intuitionistic logic revisited. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2000, St Andrews, Scotland, UK, July 3-7, 2000, Proceedings*, pages 252–267, 2000.

[6] Rajeev Goré, Linda Postniece, and Alwen Tiu. Cut-elimination and proof-search for bi-intuitionistic logic using nested sequents. In *Advances in Modal Logic 7, papers from the seventh conference on "Advances in Modal Logic," held in Nancy, France, 9-12 September 2008*, pages 43–66, 2008.

[7] Oleg Kiselyov. An argument against call/cc, 2012. okmij.org/ftp/continuations/against-callcc.html, Accessed 15 January 2020.

[8] Luís Pinto and Tarmo Uustalu. Proof search and counter-model construction for bi-intuitionistic propositional logic with labelled sequents. In M. Giese and A. Waaler, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 295–309, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[9] Cecylia Rauszer. A formalization of the propositional calculus of H-B logic. *Studia Logica: An International Journal for Symbolic Logic*, 33(1):23–34, 1974.

[10] Cecylia Rauszer. Semi-Boolean algebras and their application to intuitionistic logic with dual operations. *Fundamenta Mathematicae LXXXIII*, pages 219–249, 1974.

[11] Michael Shulman, Toby Bartels, and Sridhar Ramesh. cocartesian closed category, 2010. https://ncatlab.org/nlab/show/cocartesian+closed+category, Accessed 15 January 2020.

[12] Luca Tranchini. Natural deduction for bi-intuitionistic logic. *Journal of Applied Logic*, 25:72–96, 2017.

12 Foundations of logic and type theory

Proof terms for generalized classical natural deduction

Herman Geuvers[1] and Tonny Hurkens

[1] Radboud University Nijmegen & Technical University Eindhoven (NL)
herman@cs.ru.nl
[2] hurkens@science.ru.nl

We build on the method of deriving natural deduction rules for a connective c from the truth table t_c of c, as it has been introduced in [1, 2]. In [1] we defined the method for both constructive logic and classical logic, and the constructive case has been studied in more detail, using proof terms for deductions, in [2, 3]. Here we focus on the classical case: we introduce proof terms for the classical natural deduction rules that we extract from a truth table and we use them to study *normal deductions*. These normal deductions, or *deductions in normal form*, should satisfy the *sub-formula property*, that is: every formula that occurs in a normal deduction is a sub-formula of the conclusion or one of the assumptions. We will prove this property by giving a *normalization procedure*, that transform a deduction into one in normal form.

A special advantage of our general method of extracting deduction rules from the truth table is that our deduction rules have a specific format. This allows us to give a generic format for the proof terms and to study them generally for an arbitrary set of connectives. Our method also has the advantage (which has already been discussed in [1]), that we can study connectives "in isolation": e.g. there are classical rules for implication, which use only implication (and no negation). In case a connective c is *monotonic*, the constructive and the classical deduction rules are equivalent, but in case c is *non-monotonic* they are not. (Connective c of arity n is monotonic iff its truth table function $t_c : \{0,1\}^n \to \{0,1\}$ is monotonic with respect to the ordering induced by $0 \leq 1$.) We will also show that, if one non-monotonic connective has classical rules, then all connectives "become" classical. So, e.g. if we have $\{\to, \neg\}$ as connectives with classical rules for $\to$ and constructive rules for $\neg$, we can derive the classical rules for $\neg$.

Let's be a bit more precise about the deduction rules and the proof terms. The elimination rules have the following form. Φ is the formula we eliminate. We have $\Phi = c(A_1, \ldots, A_n)$ where c is a connective of arity n and $n = k + \ell$. The formula D is arbitrary.

$$\frac{\vdash \Phi \quad \vdash A_{i_1} \quad \ldots \quad \vdash A_{i_k} \qquad A_{j_1} \vdash D \quad \ldots \quad A_{j_\ell} \vdash D}{\vdash D}\ \text{el}$$

So, $A_{i_1}, \ldots, A_{i_k}, A_{j_1}, \ldots, A_{j_\ell}$ are the direct sub-formulas of $\Phi = c(A_1, \ldots, A_n)$. We refer to the A_i as *lemma* and the A_j as *casus* in the derivation rule. The classical introduction rules have the following form. Again, c is a connective of arity n, $\Phi = c(A_1, \ldots, A_n)$ and $n = k + \ell$. (Of course, every rule has its own specific sequence $i_1, \ldots, i_k, j_1, \ldots j_\ell$.)

$$\frac{\Phi \vdash D \quad \vdash A_{i_1} \quad \ldots \quad \vdash A_{i_k} \qquad A_{j_1} \vdash D \quad \ldots \quad A_{j_\ell} \vdash D}{\vdash D}\ \text{in}$$

For a concrete connective c, we derive the elimination and introduction rules from the truth table, as described in [1, 2], where every line in the truth table t_c gives a deductions rule: an elimination rule if $t_c(a_1, \ldots, a_n) = 0$ and an introduction rule if $t_c(a_1, \ldots, a_n) = 1$.

Given a logic with classical derivation rules as derived from truth tables for a set of connectives $\mathcal{C}$, we can define the typed λ-calculus $\lambda^{\mathcal{C}}$, which has judgments $\Gamma \vdash t : A$, where A is a formula, Γ is a set of declarations $\{x_1 : A_1, \ldots, x_m : A_m\}$, where the A_i are formulas and the

x_i are term-variables such that every x_i occurs at most once in Γ, and t is a *proof-term*. The abstract syntax for proof-terms, Term, is as follows.

$$t ::= x \mid (\lambda y : A.t) \cdot \{\overline{t} \,;\, \overline{\lambda x : A.t}\} \mid t \cdot [\overline{t} \,;\, \overline{\lambda x : A.t}]$$

where x ranges over variables.

The terms are *typed* using the following derivation rules, where the first rule is the *axiom* rule basically stating that $\Gamma \vdash A$ if $A \in \Gamma$.

$$\frac{}{\Gamma \vdash x_i : A_i} \text{ if } x_i : A_i \in \Gamma$$

$$\frac{\Gamma \vdash t : \Phi \quad \ldots \quad \Gamma \vdash p_k : A_k \quad \ldots \quad \Gamma, y_\ell : A_\ell \vdash q_\ell : D \quad \ldots}{\Gamma \vdash t \cdot [\overline{p} \,;\, \overline{\lambda y : A.q}] : D} \text{ el}$$

$$\frac{\Gamma, z : \Phi \vdash t : D \quad \ldots \quad \Gamma \vdash p_i : A_i \quad \ldots \quad \Gamma, y_j : A_j \vdash q_j : D \quad \ldots}{\Gamma \vdash (\lambda z : \Phi.t) \cdot \{\overline{p} \,;\, \overline{\lambda y : A.q}\} : D} \text{ in}$$

Here, $\overline{p}$ is the sequence of terms $p_1, \ldots, p_{m'}$ for all the 1-entries in the truth table, and $\overline{\lambda y : A.q}$ is the sequence of terms $\lambda y_1 : A_1.q_1, \ldots, \lambda y_m : A_m.q_m$ for all the 0-entries in the truth table.

To reduce the proof terms (and thereby the deductions) to normal form, we first perform *permutation reductions* and then we eliminate *detours*. This is similar to the constructive case, except for now

- a term is in *permutation normal form* if all lemmas are variables,
- a *detour* is an elimination of Φ followed by an introduction of Φ.

Note the difference with constructive logic, where a detour is an introduction directly followed by an elimination. Here it is the other way around, and the introduction need not follow the elimination directly.

We can be more precise by giving the following abstract syntax N for *permutation normal forms*:

$$N ::= x \mid (\lambda y : A.N) \cdot \{\overline{z} \,;\, \overline{\lambda x : A.N}\} \mid y \cdot [\overline{z} \,;\, \overline{\lambda x : A.N}],$$

where x, y, z range over variables. We can obtain a deduction in permutation normal form by moving applications of an elimination or introduction rule that have a non-trivial lemma upwards, until all lemmas become trivial: the proof-terms are variables. (This only works for the classical case!) Now, a *detour* is a pattern of the following shape

$$(\lambda x : \Phi. \ldots (x \cdot [\overline{v} \,;\, \overline{\lambda w : B.s}]) \ldots) \cdot \{\overline{z} \,;\, \overline{\lambda y : A.q}\}$$

that is, an elimination of Φ followed by an introduction of Φ, with an arbitrary number of steps in between. For terms in permutation normal form, we show how detours can be eliminated, obtaining a term in normal form which satisfies the sub-formula property. It should be noted that in the above case, this need not be the only occurrence of x, so this elimination is not straightforward. Another situation is that x may not ocuur at all; that is the simplest situation and the sub-term $(\lambda x : \Phi.t) \cdot \{\overline{z} \,;\, \overline{\lambda y : A.q}\}$ can simply be replaced by t.

References

[1] H. Geuvers and T. Hurkens. Deriving natural deduction rules from truth tables. In *ICLA*, volume 10119 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2017.

[2] H. Geuvers and T. Hurkens. Proof Terms for Generalized Natural Deduction. In A. Abel, F. Nordvall Forsberg, and A. Kaposi, editors, *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*, volume 104 of *LIPIcs*, pages 3:1–3:39, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] H. Geuvers, I. van der Giessen, and T. Hurkens. Strong normalization for truth table natural deduction. *Fundam. Inform.*, 170(1-3):139–176, 2019.

Did Palmgren Solve a Revised Hilbert's Program?

Anton Setzer

Dept. of Computer Science, Swansea University Bay Campus, Fabian Way, Swansea SA1 8EN, UK
a.g.setzer@swansea.ac.uk

In Memory of Erik Palmgren

Abstract

We revisit the article by Palmgren giving an embedding of iterated inductive definitions into Martin-Löf Type Theory, and explain in what sense it provides an early substantial solution to a revised Hilbert's program.

Palmgren's result didn't provide a sharp lower bound. We present a restricted version of Martin-Löf Type Theory with W-type and one universe, for which the embedding of Palmgren works as well and for which Palmgren's lower bound is sharp. We give a proof sketch for the sharpness of this bound.

1 Palmgren's Result and a Revised Hilbert's Program

On the UNESCO logic day 2020, the author gave a talk in Swansea dedicated to the memory of Erik Palmgren, who died recently unexpectedly at young age. Apart from [4] we presented the article [3], which gave an embedding of the theory of intuitionistic strictly positive iterated inductive definitions $\mathrm{ID}_{<\epsilon_0}$ into the theory $\mathrm{ML}_1\mathrm{W}$ of Martin-Löf Type Theory (MLTT) with W-type and one Universe. Regarding the proof theoretic strength of $\mathrm{ID}_{<\epsilon_0}$ it is known (see [1] or [5]), that $|\mathrm{ID}_{<\epsilon_0}| = \psi_{\Omega_1}(\Omega_{\epsilon_0}) > \psi_{\Omega_1}(\Omega_\omega) = |(\Pi^1_1 - \mathrm{CA})_0|$. Therefore combining Palmgren's result with this proof theoretic result it follows that $\mathrm{ML}_1\mathrm{W}$ proves the consistency of $(\Pi^1_1-\mathrm{CA})_0$. The latter theory is the strongest of the big five theories of reverse mathematics [9], and it has been observed in reverse mathematics that most real mathematical theorems can be shown in one of these big five theories [9].

MLTT has been developed in such a way that we can get as much as possible a direct insight into the validity of the judgements provable in it. Of course there are limitations – you cannot get around Gödel's 2nd Incompleteness Theorem, and there is no absolute consistency proof of any theory. However, MLTT is the best approximation to a self evident theory, and can therefore be considered as a good candidate to an extensions of finitary methods in a revised Hilbert's program. So if we consider MLTT as a version of extended finitary methods, and combine Palmgren's result with the results of proof theory and reverse mathematics, we obtain a proof using extended finitary methods of the consistency of a theory which allows to formalise a substantial part of mathematics. Therefore we obtain a substantial solution to a revised Hilbert's program.

2 Palmgren's Result as a Sharp Bound

Palmgren already observed that his result is not sharp and conjectured that autonomous iterated inductive definitions can be interpreted into $\mathrm{ML}_1\mathrm{W}$. The author showed in his PhD thesis [6, 7, 8], that the strength of $\mathrm{ML}_1\mathrm{W}$ is much stronger, namely $\psi_{\Omega_1}(\Omega_{\mathrm{I}+\omega})$. When revisiting Palmgren's result it became clear in what sense Palmgren wasn't using the full power of $\mathrm{ML}_1\mathrm{W}$. Palmgren didn't make use of any W-type build on top of U. Furthermore, he made use of

induction on Set-level of the $\mathbb{N}$, but only of induction over W restricted to elements in U. So we can replace the target theory $\mathrm{ML_1W}$ used by Palmgren by a weaker theory $\mathrm{ML_1W^-}$ which restricts the W-type to U. With this restriction Palmgren's embedding of $\mathrm{ID}_{<\epsilon_0}$ works as well, and we even obtain a sharp bound (details still need to be scrutinised).

Definition. *We assume the small logical framework, i.e. the logical framework restricted to elements of* Set*. The theory* $\mathrm{ML_1W^-}$ *is obtained from* $\mathrm{ML_1W}$ *by omitting the rules for the* W*-type and closure of* U *under* $\widehat{\mathrm{W}}$ *and adding the following rules:*

- *Formation rule for* W *restricted to* U*:*

$$\frac{a : \mathrm{U} \qquad b : \mathrm{T}\ a \to \mathrm{U}}{\mathrm{W}\ a\ b : \mathrm{Set}}$$

- *Introduction rule for* W*:*

$$\sup{}_{a,b} : (r : \mathrm{T}\ a)(s : \mathrm{T}\ (b\ c) \to \mathrm{W}\ a\ b) \to \mathrm{W}\ a\ b$$

- *Elimination rule for* W *restricted to elements of* U*:*

$$\frac{\begin{array}{c} c : \mathrm{W}\ a\ b \to \mathrm{U} \\ \mathit{step} : (x : \mathrm{T}\ a)(y : \mathrm{T}\ (b\ x) \to \mathrm{W}\ a\ b)(\mathit{ih} : (u : \mathrm{T}\ (b\ x)) \to \mathrm{T}\ (c\ (y\ u))) \to \mathrm{T}\ (c\ (\sup{}_{a,b}\ x\ y)) \end{array}}{\mathrm{Elim_W}\ a\ b\ c\ \mathit{step} : (d : \mathrm{W}\ a\ b) \to \mathrm{T}\ (c\ d)}$$

- *Equality rule for* W*:*

$$\mathrm{Elim_W}\ a\ b\ c\ \mathit{step}\ (\sup{}_{a,b}\ d\ e) = \mathit{step}\ d\ e\ ((u)\mathrm{Elim_W}\ a\ b\ c\ \mathit{step}\ (e\ u))$$

- *Closure of* U *under* $\widehat{\mathrm{W}}$*:*

$$\widehat{\mathrm{W}} : (a : \mathrm{U})(b : \mathrm{T}\ a \to \mathrm{U}) \to \mathrm{U} \qquad\qquad \mathrm{T}\ (\widehat{\mathrm{W}}\ a\ b) = \mathrm{W}\ a\ b : \mathrm{Set}$$

Theorem. $|\mathrm{ML_1W^-}| = |\mathrm{ID}_{<\epsilon_0}| = \psi_{\Omega_1}(\Omega_{\epsilon_0})$.

Proof Sketch (needs to be scrutinised).

For the result $|\mathrm{ID}_{<\epsilon_0}| = \psi_{\Omega_1}(\Omega_{\epsilon_0})$ see [1] and in modern notation [5], p. 332.

Lower bound $|\mathrm{ML_1W^-}| \geq |\mathrm{ID}_{<\epsilon_0}|$**:** The interpretation of Palmgren can be easily adapted to interpret $\mathrm{ID}_{<\epsilon_0}$ into $\mathrm{ML_1W^-}$: Since we have the full elimination rules for $\mathbb{N}$ we can iterate any operator $\mathbb{N} \to (\mathbb{N} \to \mathrm{U}) \to (\mathbb{N} \to \mathrm{U})$ up to β times for any $\beta < \epsilon_0$ by using the same application of Gentzen's trick as used by Erik Palmgren.

Each step of the inductive definition can be interpreted as an element $\mathbb{N} \to \mathrm{U}$ by using the restricted W-type. However, in $\mathrm{ML_1W^-}$ we obtain only induction into elements of U for the interpreted inductive definitions. This is sufficient, since induction in $\mathrm{ID}_{<\epsilon_0}$ is restricted to formulas in this language, which can be represented as elements of U.

Upper bound $|\mathrm{ML_1W^-}| \leq \psi_{\Omega_1}(\Omega_{\epsilon_0})$**:** By [2] Theorem 8.4 (more modern notation in [5], p. 332), $|\mathrm{W-KPI}| = \psi_{\Omega_1}(\Omega_{\epsilon_0})$ where $(\mathrm{W-KPI})$ is KPI with full induction scheme $\mathrm{IND}_{\mathbb{N}}$ over $\mathbb{N}$ but foundation replaced by the foundation axiom $\mathrm{I}_{\in}$. We can build a PER model of $\mathrm{ML_1W^-}$ in $(\mathrm{W-KPI})$ similarly as in [6, 8]. U, T are interpreted as the union of U^α, T^α over set theoretic ordinals α. Using $\mathrm{I}_{\in}$ we can show that $\mathrm{U}^\alpha, \mathrm{T}^\alpha$ are accumulative and form PERs, and that the interpretation of U, T is closed under the introduction rules for U. The elimination rules for W can be modelled using $\mathrm{I}_{\in}$. The elimination rule for $\mathbb{N}$ can be shown using $\mathrm{IND}_{\mathbb{N}}$.

References

[1] Wilfried Buchholz, Solomon Feferman, Wolfram Pohlers, and Wilfried Sieg. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*, volume 897 of *Lecture Notes in Mathematics*. Springer, 1981.

[2] Gerhard Jäger. *Theories for Admissible Sets: A Unifying Approach to Proof Theory*. Studies in Proof Theory Lecture Notes, Vol 2. Bibiopolois, Naples, 1987.

[3] Erik Palmgren. Type-theoretic interpretation of iterated, strictly positive inductive definitions. *Archive for Mathematical Logic*, 32:75–99, 1992.

[4] Erik Palmgren. On universes in type theory. In G. Sambin and J.M. Smith, editors, *Twenty-five years of constructive type theory: proceedings of a congress held in Venice, October 1995*, volume 36, pages 191 – 204. Oxford University Press, 1998.

[5] Wolfram Pohlers. Chapter 4: Subsystems of set theory and second order number theory. In Samuel R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 209 – 335. Elsevier, 1998.

[6] A. Setzer. *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe*. PhD thesis, Mathematisches Institut, Universität München, Munich, Germany, 1993. Available from http://www.cs.swan.ac.uk/~csetzer/articles/weor0.pdf.

[7] Anton Setzer. Well-ordering proofs for Martin-Löf type theory. *Annals of Pure and Applied Logic*, 92:113 – 159, 1998. https://doi.org/10.1016/S0168-0072(97)00078-X.

[8] Anton Setzer. An upper bound for the proof theoretic strength of Martin-löf type theory with W-type and one universe, 2020. To appear in: Reinhard Kahle, Michael Rathjen (Eds): Festschrift on the occasion of Schütte's 111th Birthday. Springer.

[9] Stephen G Simpson. *Subsystems of second order arithmetic*, volume 1. Cambridge University Press, 2nd edition, 2009.

Axiom C and Genericity for System F

Sergei Soloviev[1]

IRIT, Paul Sabatier University, Toulouse, France,
soloviev@irit.fr

In a joint paper [1] published in 1993 we presented a genericity theorem for system F with a supplementary equality axiom, called Axiom C:

(**Axiom C**) $M\tau = M\tau'$ for $\Gamma \vdash M : \forall X.\sigma$ and $X \notin FV(\sigma)$

The modified system was called F_c and the main theorem was formulated as follows:

Theorem 9.3 (Genericity) *Let* $\Gamma \vdash M, N : \forall X.\sigma$. *If* $M\tau =_{F_c} N\tau$ *for some type* τ, *then* $M =_{F_c} N$. (Notice that the condition $X \notin FV(\sigma)$ is *not* included.)

We wrote in [1] "The Genericity Theorem states ... that, in F_c, if two second-order terms coincide on an input type, then they are, in fact, the same function..."

More recently Ralph Matthes pointed out (to this author) that certain formulations in [1] seem not very precise and as a consequence some proofs seem not sufficiently clear as well. Ralph's questions mostly concerned the treatment of contexts and related questions such as the variable convention and the eigenvariable condition. As his PhD thesis [2] defended in 1999 shows (see especially section 2.1), there are many subtle aspects that were not fully realized at the time when our own paper was written (1992).

This talk will summarize main answers/comments and the results of supplementary exploration of the question of genericity (and term equality) in F_c that may be of interest to "Types" community.

First of all, Theorem 9.3 in its original formulation remains valid (though some comments are due). A more rigourous presentation of F and F_c than in [1] will be given (taking into account the progress in the study of higher order and dependent type systems). In [1] to some extent we mixed the "Church-style" (types given to free variables within terms) and "Curry-style" presentation (contexts to define the types of free variables, and typing judgements for terms). Matthes [2] used essentially the Church-style. In this talk the presentation using typing judgements is preferred because it helps a) to underline some interesting properties of C-equality and b) to make comparison with dependent type systems.

Type and term variables belong to (countably infinite) disjoint sets of symbols. We use $X, Y, Z...$ and $x, y, z...$ for type and term variables respectively, $\rho, \sigma, \tau...$ for types and $M, N, ...$ for terms. The syntax of types and (raw) terms is standard:

$$Types\ \ \sigma ::= X \,|\, \sigma \to \tau \,|\, \forall X.\sigma$$

$$Terms\ \ M ::= x \,|\lambda x : \sigma.M \,|\, MN \,|\, \Lambda X.M \,|\, M\tau$$

Two-part contexts $\Delta;\Gamma$ are considered where Δ is a finite set of type variables, and Γ a finite set of type assignments of the form $x : \sigma$ (x being distinct term variables). The syntactic definitions above permit already to define the sets of free variables $FV(\sigma)$, $FV(M)$, $FV(\Gamma)$ (binders are $\forall$, λ and Λ, and these sets include both type and term variables).

To the ordinary typing judgements $\Delta;\Gamma \vdash M : \sigma$ we add the equality judgements of the form $\Delta;\Gamma \vdash M = N : \sigma$. (All judgements are considered up to α-equality that is seen as a meta-level relation.) The rules are modified (updated) accordingly. For example:

$$\frac{\Delta;\Gamma \vdash M : \sigma\ (X \in \Delta, X \notin FV(\Gamma))}{\Delta\backslash\{X\};\Gamma \vdash \Lambda X.M : \forall X.\sigma}(\forall - int)\quad \frac{\Delta;\Gamma \vdash \Lambda X.(MX) : \forall X.\sigma\ (X \notin FV(M))}{\Delta,\Gamma \vdash \Lambda X.(MX) = M : \forall X.\sigma}(\Lambda - int)$$

In $\Lambda - int$ the derivability of the premise will exclude the possibility of X being present in the types of free term variables of M.

Axiom C is represented by the following equality rule:

$$\frac{\Delta;\Gamma \vdash M : \forall X.\sigma \ (X \notin FV(\sigma), \tau, \rho \in Types, FV(\tau), FV(\rho) \subseteq \Delta)}{\Delta;\Gamma \vdash M\tau = M\rho : \sigma}(C).$$

Notice that axiom C does not change typing and the class of well-typed terms in F.

Naturally, if $\Delta;\Gamma \vdash M : \sigma$ and we change the context, the type of M may change. However the $\beta\eta$-equality is invariant w.r.t. the change of context in the following sense ($=_{\beta\eta}$ means that the rule C is not used in the derivation of this equality):

Proposition 1. Let $\Delta;\Gamma \vdash M : \sigma$, $\Delta;\Gamma \vdash N : \sigma$ and $\Delta;\Gamma \vdash M =_{\beta\eta} N : \sigma$. If (in another context) $\Delta_1;\Gamma_1 \vdash M : \tau$ and $\Delta_1;\Gamma_1 \vdash N : \tau$ then $\Delta_1;\Gamma_1' \vdash M =_{\beta\eta} N : \tau$.

For C-equality the situation is different. It *may* depend on context. This point went unnoticed in [1], probably because of some confusion of "Church" and "Curry"-styles. Let $\bot = \forall X.X, \top = \forall X.(X \to X)$.

Example 1. Let $\Gamma = x : \bot, \Gamma_1 = x : \forall Y.\forall Z.(\bot \to \bot)$, $M = x\top\bot, N = x\bot(\bot \to \bot)$. The Δ part is empty, we have $\Gamma \vdash M, N : \bot \to \bot$ and $\Gamma_1 \vdash M, N : \bot \to \bot$ (the terms even have the same types). However with Γ the axiom C cannot be applied at all, and with Γ_1 it can (twice), so we have $\Gamma_1 \vdash M =_{F_c} N : \bot \to \bot$ but $\Gamma \vdash M \neq_{F_c} N : \bot \to \bot$.

It turns out that in theorem 9.3 cited above the role of context is crucial. That is, in one context we may have $\Delta;\Gamma \vdash M\rho =_{F_c} N\rho : [\rho/X]\sigma$ for some type ρ, and thus $\Delta;\Gamma \vdash M =_{F_c} N : \forall X.\sigma$ (genericity), but in another just $M\rho \neq_{F_c} N\rho$. Appropriate examples may be easily obained from the example above:

Example 2. Let $M' = \Lambda U.x\top U, N' = \Lambda U.x\bot(U \to U)$ with $\rho = \bot$ and the same contexts Γ and Γ_1 as in example 1.

It means that the claim "if two second-order terms coincide on an input type, then they are, in fact, the same function..." has to be corrected. The situation is much more interesting: these terms (seen as functions with parameters taken from context) may represent the same (constant) function on types for some contexts and different functions for others. This behaviour has certainly to be explored further.

The talk will present also the list of minor corrections that are necessary in [1] (they concern mostly the formulation of some lemmas, technically the proofs remain almost unchanged).

References

[1] Longo, G., Milsted, K., and Soloviev, S. (1993) The Genericity Theorem and effective Parametricity in Polymorphic lambda-calculus. *Theoretical Computer Science* **121**, 323-349.

[2] Matthes, R. Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types. PhD Thesis, Ludwig-Maximilian University, Munich, defended on 2 February 1999.

Towards Completeness of Full Simply Typed Lambda Calculus

Silvia Ghilezan[1,2] and Simona Kašterović[1]

[1] Faculty of Technical Sciences, University of Novi Sad
Novi Sad, Serbia
{gsilvia, simona.k}@uns.ac.rs
[2] Mathematical Institute SASA
Belgrade, Serbia

The full simply typed lambda calculus is the simply typed lambda calculus extended with product type and sum type. We briefly recall the grammar which generates terms and types. Basic notions and definitions can be found in [1, 4, 6, 7]. Term expressions, aka terms, are generated by the following grammar:

$$\begin{aligned} M, N ::= & x | \lambda x.M | MN | \langle M, N \rangle | \pi_1(M) | \pi_2(M) \\ & | \text{in}_1(M) | \text{in}_2(M) | \text{case } M \text{ of } \text{in}_1(x) \Rightarrow N \mid \text{ in}_2(y) \Rightarrow L | \langle\rangle | \mathsf{abort}(M) \end{aligned}$$

where x is a term-variable. Types are generated by the following grammar:

$$\sigma, \tau ::= a \mid \sigma \to \tau \mid \sigma \times \tau \mid \sigma + \tau \mid 0 \mid 1$$

where a is a type variable. The type assignment system with simple (functional), product and sum types is denoted by $\Lambda^{\to,\times,+}$.

In this paper, we propose a Kripke-style semantics for full simply typed lambda calculus. For the lack of space we will just briefly give the idea we use for defining semantics combining the approcaches in [4] and [5]. In [4] the author did not consider Kripke-style semantics, however the notions of applicative structure, extensionality and combinators were defined similarly. On the other hand, in [5] the authors considered Kripke lambda models for simply typed lambda calculus (without product and sum type). The semantics proposed in [5] and the semantics we propose are different. In [5] the authors defined the meaning of well-typed terms, i.e. $[\![\Gamma \vdash M : \sigma]\!]$. Our aim is to define a semantics that allows to define the meaning of a term $[\![M]\!]$ not considering its type. The model will satisfy that a term has a certain type if the value which represents the meaning of the term belongs to the appropriate set.

Definition 1. *A Kripke applicative structure* for $\Lambda^{\to,\times,+}$ is a tuple $\mathcal{K} = \langle W, \leq, \{D_w\}, \{A^\sigma_w\}, \{App_w\}, \{Proj_{1,w}\}, \{Proj_{2,w}\}, \{Inl_w\}, \{Inr_w\}, \{i_{w,w'}\}\rangle$, which consists of:

(i) a set W of *possible worlds* partially ordered by $\leq$,

(ii) a family $\{D_w\}$ of sets, indexed by worlds w, the set D_w represents a domain of the world w, $\{D_w\}$ is a short for $\{D_w\}_{w\in W}$ and the same applies to other families,

(iii) a family $\{A^\sigma_w\}$ of sets indexed by types σ and worlds w, which satisfies the following:

- for all $w \in W$, for all $\sigma \in \mathsf{Type}$, $A^\sigma_w \subseteq D_w$, A^0_w is empty, i.e. $A^0_w = \emptyset$, and A^1_w has one element, i.e. $A^1_w = \{1_w\}$, $1_w \in D_w$,
- there exists an injection function $H : D_w \uplus D_w \to D_w$, such that for all $\sigma, \tau \in \mathsf{Type}$, $H : A^\sigma_w \uplus A^\tau_w \to A^{\sigma+\tau}_w$,

- there exists an injection function $G : D_w \to D_w \times D_w$ such that for all $\sigma, \tau \in \mathsf{Type}$, $G : A_w^{\sigma\times\tau} \to A_w^\sigma \times A_w^\tau$,

(iv) a family $\{App_w\}$ of *application functions* $App_w : D_w \times D_w \to D_w$ indexed by worlds w, such that for all $\sigma, \tau \in \mathsf{Type}$, $App_w \restriction (A_w^{\sigma\to\tau} \times A_w^\sigma) : A_w^{\sigma\to\tau} \times A_w^\sigma \to A_w^\tau$,

(v) a family $\{Proj_{1,w}\}$ of *first projection functions* $Proj_{1,w} : D_w \to D_w$ indexed by worlds w, such that for all $\sigma, \tau \in \mathsf{Type}$, $Proj_{1,w} \restriction A_w^{\sigma\times\tau} : A_w^{\sigma\times\tau} \to A_w^\sigma$,

(vi) a family $\{Proj_{2,w}\}$ of *second projection functions* $Proj_{2,w} : D_w \to D_w$ indexed by worlds w, such that for all $\sigma, \tau \in \mathsf{Type}$, $Proj_{2,w} \restriction A_w^{\sigma\times\tau} : A_w^{\sigma\times\tau} \to A_w^\tau$,

(vii) a family $\{Inl_w\}$ of *left injection functions* $Inl_w : D_w \to D_w$ indexed by worlds w, such that for all $\sigma, \tau \in \mathsf{Type}$, $Inl_w : A_w^\sigma \to A_w^{\sigma+\tau}$,

(viii) a family $\{Inr_w\}$ of *right injection functions* $Inr_w : D_w \to D_w$ indexed by worlds w, such that for all $\sigma, \tau \in \mathsf{Type}$, $Inr_w : A_w^\tau \to A_w^{\sigma+\tau}$,

(ix) a family $\{i_{w,w'}\}$ of *transition functions* $i_{w,w'} : D_w \to D_{w'}$ indexed by pairs of worlds $w \leq w'$, such that for all $\sigma \in \mathsf{Type}$, $i_{w,w'} \restriction A_w^\sigma : A_w^\sigma \to A_{w'}^\sigma$ and all transition functions satisfies the following conditions:

$$i_{w,w} : D_w \to D_w \text{ is the identity}$$
$$i_{w',w''} \circ i_{w,w'} = i_{w,w''} \text{ for all } w \leq w' \leq w''$$

We also require that the application functions, the projection functions and the injection functions commute with the transition in a natural way:

$$(\forall f \in D_w)\ (\forall a \in D_w)\ (\forall w' \in W, w \leq w') \qquad \begin{aligned} i_{w,w'}(App_w(f,a)) &= App_{w'}(i_{w,w'}(f), i_{w,w'}(a)) \\ i_{w,w'}(Proj_{1,w}(a)) &= Proj_{1,w'}(i_{w,w'}(a)) \\ i_{w,w'}(Proj_{2,w}(a)) &= Proj_{2,w'}(i_{w,w'}(a)) \\ i_{w,w'}(Inl_w(a)) &= Inl_{w'}(i_{w,w'}(a)) \\ i_{w,w'}(Inr_w(a)) &= Inr_{w'}(i_{w,w'}(a)) \end{aligned}$$

We are only interested in applicative structures which are extensional and have combinators (for more details see [4]). The lambda calculus we consider is related to full intuitionistic propositional logic via the Curry-Howard correspondence ([2]). The existence of the elements, called combinators, in a model is motivated by this correspondence. More precisely, the axioms of the Hilbert-style system for intuitionistic propositional logic were the motivation for defining combinators as elements of models. We have used the translation of lambda terms into combinatory terms ([3, 4]) in order to prove that the meaning of terms is well-defined.

We have proved that the type assignment system $\Lambda^{\to,\times,+}$ is sound with respect to the proposed semantics. Our goal is to prove completeness of the type assignment system $\Lambda^{\to,\times,+}$. The idea is to use the notion of a canonical model for proving completeness.

References

[1] Roberto Di Cosmo and Delia Kesner. A confluent reduction for the extensional typed lambda-calculus with pairs, sums, recursion and terminal object. In *Automata, Languages and Programming, 20nd International Colloquium, ICALP93, Lund, Sweden, July 5-9, 1993, Proceedings*, pages 645–656, 1993.

[2] William A. Howard. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. London : Academic Press, 1980 (originally circulated 1969).

[3] Albert R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, 1982.

[4] John C. Mitchell. *Foundations for programming languages.* Foundation of computing series. MIT Press, 1996.

[5] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):99–124, 1991.

[6] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations.* Software Foundations series, volume 2. Electronic textbook, May 2018. Version 5.5.

[7] Gabriel Scherer. Deciding equivalence with sums and the empty type. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 374–386, 2017.

Extended Abstract: Principal types are game strategies

Furio Honsell, Marina Lenisa, Marino Miculan, and Ivan Scagnetto

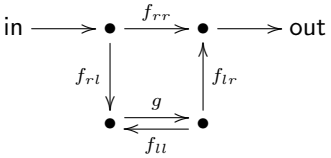
Università di Udine, Italy
{furio.honsell, marina.lenisa, marino.miculan, ivan.scagnetto}@uniud.it

In this paper, the authors pursue the *explanation* of *Geometry of Interaction (GoI) semantics* in terms of *principal types*. This line of research initiated in [7], where some of the authors put forward a new understanding of *involutions* in Abramsky's model of reversible computation [1]. Namely, the involution interpreting a *linear λ-term* is just a notation for its *principal type* and moreover the *GoI-application* between interpretations (*execution formula*) amounts to *resolution* between principal types. This enabled us to give a partial answer to the problem, raised by Abramsky in [1], of characterizing the involutions which are interpretations of combinators. In [7], we termed the above correspondence the *strategies-as-principal types/application-as-resolution analogy*. This analogy was further extended in [8, 9] to the $\lambda!$-calculus, which extends the *linear λ-calculus* with a modal "!" operator, and subsumes full λ-calculus.

We outline a general *programme*, implicitly suggested in [2, 1, 3, 12, 13, 7, 9, 8], whereby, given a suitable *type discipline* for a (possibly modal) λ-calculus, first one defines an appropriate set of combinators where such a fragment can be faithfully compiled. Next, the *principal types* for these combinators are spelled out in the *language of moves* in a suitable G.o.I. model, thus providing an interpretation to the combinators via the *resolution-as-G.o.I.-application* analogy. Finally, principal types are extended to the full modal λ-calculus, providing an interpretation to all λ-terms. From a logical viewpoint, this programme amounts to explaining via a *Hilbert*-style logic what otherwise is given in terms of natural deduction or proof-nets. Clearly, such models are not λ-algebras in general.

For more generality, we extend the model $\mathcal{P}$ in [1] from partial involutions to *partial symmetric relations (PSR)* as follows:

Definition 1 (The Model of Partial Symmetric Relations, $\mathcal{R}$)**.**
(i) T_Σ, the language of moves*, is defined by the signature $\Sigma_0 = \{\epsilon\}$, $\Sigma_1 = \{l,r\}$, $\Sigma_2 = \{<\ ,\ >\}$ (where Σ_i is the set of constructors of arity i); terms $r(x)$ are* output words*, while terms $l(x)$ are* input words *(often simply denoted by rx and lx);*
(ii) $\mathcal{R}$ is the set of partial symmetric relations *over T_Σ;*
(iii) the operation of replication *is defined by $!f = \{(<t,u>,<t,v>) \mid t \in T_\Sigma \wedge (u,v) \in f\}$;*
(iv) the notion of linear application *is defined by $f \cdot g = f_{rr} \cup (f_{rl}; g; (f_{ll}; g)^*; f_{lr})$, where $f_{ij} = \{(u,v) | (i(u), j(v)) \in f\}$, for $i,j \in \{r,l\}$ (see diagram below), where ";" denotes postfix composition.*



We make use of a modal extension of linear λ-calculus, and use a rigid form of *intersection*-type discipline (see [6]) to control multiple occurrences of bound variables. Namely we put:

Definition 2 (The $\lambda!$-calculus and its $! - \wedge$ -type discipline)**.**

- $(\Lambda^! \ni)\ M ::= x \mid MM \mid \lambda x.M \mid !M \mid \lambda !x.M$

- ($Type^! \ni$) $\tau, \sigma ::= \alpha \mid \tau \multimap \sigma \mid \tau \to \sigma \mid\ !\tau \mid\ !_u\tau \mid \tau \wedge \sigma$, *where* α *denotes a type variable in* $TVar$, *and* $u, v \in T_\Sigma[IVar]$ *denote index terms over a set* $IVar$ *of index variables.*

There are two natural inverse morphisms, one from types to relations and one from (coherent) relations to types, namely $\mathcal{M}_{T2R} : Type^! \to \mathcal{R}$ and $\mathcal{M}_{R2T} : \mathcal{R} \to Type^!$. We illustrate these by way of a few examples concerning the combinators B, W^-, W, the pairs in the relations denote the positions of the occurrences of the same type variable or the same subtype in the type-tree (see [9] for more details):

$$\begin{array}{lcll}
B & : & (\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta & BMNP = M(NP) \\
W^- & : & (\alpha \to \alpha \to \beta) \to \alpha \to \beta & W^-MN = MNN \\
W & : & (!_i\alpha \multimap !_j\beta \multimap \gamma) \multimap (!_{li}\alpha \wedge !_{rj}\beta) \multimap \gamma & \mathrm{WM!N} = M!N!N. \\
\mathcal{M}_{T2R}((\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta) & = & \{r^3x \leftrightarrow lrx, l^2x \leftrightarrow rlrx, rl^2x \leftrightarrow r^2lx\} & \\
\mathcal{M}_{T2R}((\alpha \to \alpha \to \beta) \to \alpha \to \beta) & = & \{r^2x \leftrightarrow lr^2x, l^2x \leftrightarrow lrlx, lrlx \leftrightarrow rlx, & \\
 & & \qquad lrx \leftrightarrow l^2x\} & \\
\mathcal{M}_{T2R}((!_i\alpha \multimap !_j\beta \multimap \gamma) \multimap (!_{li}\alpha \wedge !_{rj}\beta) \multimap \gamma) & = & \{r^2x \leftrightarrow lr^2x, l^2\langle x, y\rangle \leftrightarrow rl\langle lx, y\rangle, & \\
 & & \qquad lrl\langle x, y\rangle \leftrightarrow rl\langle rx, y\rangle\}. &
\end{array}$$

We formalize first the relation between type resolution and GoI application of partial symmetric relations. Given two types $\sigma_1 \to \sigma_2$ and τ such that S is the unification of σ_1 and τ, we have $\mathcal{M}_{T2R}(S(\sigma_2)) = \mathcal{M}_{T2R}(S(\sigma_1 \to \sigma_2)) \cdot \mathcal{M}_{T2R}(S(\tau))$. The reverse needs not hold, in that GoI application can be non-empty even if the types do not resolve. For this to hold we need to resolve instead suitable greatest lower bounds w.r.t. a $\leq$ relation of the two types.

We show also that *Hindley-Milner* principal types for the standard simply typed λ-calculus can be viewed as particular *partial symmetric relations* over $T_\Sigma[IVar]$. Namely we prove:

Proposition 1 (Hindley-Milner Principal Types)**.**

- *The interpretation of a simply typed closed* λ*-term* M *in* R *is the PSR corresponding to its Hindley-Milner principal type. Conversely, the type corresponding to the PSR interpreting a closed typable term* M *is its principal type. Namely, if* $\vdash_{HM} M : \tau$, *then* $[\![M]\!]^{\mathcal{R}} = \mathcal{M}_{T2R}(\tau)$, *and conversely* $\vdash_{HM} M : \mathcal{M}_{R2T}([\![M]\!]^{\mathcal{R}})$.

- *Moreover if* M, N, MN *are typable with simple types, then* $\mathcal{M}_{R2T}([\![M]\!]^{\mathcal{R}} \cdot^{\mathcal{R}} [\![N]\!]^{\mathcal{R}})$ *is the principal type of* MN, *and conversely the relation corresponding to the principal type of their application is its interpretation.*

In the full paper we explore how to apply the programme outlined above to the *elementary affine* λ*-calculus* [18, 20, 11, 4, 10, 5] and the *linear light affine* λ*-calculus*, [18, 20].

In conclusion, we briefly relate the corresponding set theories to *Fitch-Prawitz* Set Theory and its models, see [19, 16, 18, 14, 20, 5].

In order to check formally computations using partial involutions, we have implemented in Erlang [15]: application of involutions, compilation of λ-terms as combinators and as involutions, and compilation of involutions as principal types and vice versa. The Web Appendix [21] includes the Erlang code. The choice of Erlang as a programming language was motivated by its powerful pattern matching features, its fast execution performances and especially its scalability in running in parallel the unification tasks generated by the massive set of rewriting rules.

References

[1] S. Abramsky. A Structural Approach to Reversible Computation. *Theoretical Computer Science* **347**(3), 2005.

[2] S. Abramsky, E. Haghverdi, P. Scott. Geometry of Interaction and linear combinatory algebras. *Mathematical Structures in Computer Science* **12**, 625–665, 2002.

[3] S. Abramsky, M. Lenisa. Linear realizability and full completeness for typed lambda-calculi. *Ann. Pure Appl. Logic* **134**(2-3), 2005.

[4] P. Baillot, K. Terui. A Feasible Algorithm for Typing in Elementary Affine Logic, *Typed Lambda Calculi and Applications (TLCA 2005)*, 55–70, 2005.

[5] P. Baillot, E. De Benedetti, S. Ronchi Della Rocca. Characterizing polynomial and exponential complexity classes in elementary λ-calculus, *Inf. and Comp.* **261**, 55–77, 2018.

[6] H. Barendregt, M. Coppo, M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment, *J. Symb. Log.* **48**(4), 1983.

[7] A. Ciaffaglione, F. Honsell, M. Lenisa, I. Scagnetto. The Involutions-as-principal types/application-as-unification Analogy, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 254–270, 2018.

[8] A. Ciaffaglione, P. Di Gianantonio, F. Honsell, M. Lenisa, I. Scagnetto. Reversible Computation and Principal Types in $\lambda!$-calculus, *The Bulletin of Symbolic Logic* **25**)(2), 931–940, 2019.

[9] A. Ciaffaglione, P. Di Gianantonio, F. Honsell, M. Lenisa, I. Scagnetto, *$\lambda!$-calculus, Intersection Types, and Involutions*, *Formal Structures for Computation and Deduction (FSCD 2019)*, LIPIcs **131**, 2019.

[10] P. Coppola, U. Dal Lago, S.Ronchi Della Rocca. *Light Logics and the Call-by-Value Lambda Calculus*, Logical Methods in Computer Science **4**(4), 2008.

[11] P. Coppola, S. Martini. Optimizing optimal reduction: A type inference algorithm for elementary affine logic, *ACM Trans. Comput. Log.* **7**(2), 219–260, 2006.

[12] P. Di Gianantonio, F. Honsell, M. Lenisa. A type assignment system for game semantics, *Theor. Comput. Sci.* **398** (1-3), 2008.

[13] P. Di Gianantonio, M. Lenisa. Innocent Game Semantics via Intersection Type Assignment Systems, *Computer Science Logic (CSL 2013)*, LIPIcs **23**, 231–247, 2013.

[14] F.Honsell, M. Lenisa, L. Liquori, I. Scagnetto. Implementing Cantor's Paradise, *Programming Languages and Systems - 14th Asian Symposium (APLAS 2016)*, 229–250, 2016.

[15] Erlang official website. http://www.erlang.org Last access: 19/01/2018.

[16] F. B. Fitch. *Symbolic logic - An Introduction*. New York, 1952.

[17] E. Haghverdi. *A Categorical Approach to Linear Logic, Geometry of Proofs and full completeness*, PhD Thesis, University of Ottawa, 2000.

[18] J.-Y. Girard. Light Linear Logic, *Information and Computation* **143**(2), 175–204, 1998.

[19] D. Prawitz. *Natural Deduction – A proof-theoretical Study*. Dover Publications, New York, 2006.

[20] K. Terui. *Light Logic and Polinomial Time Computation*, PhD thesis, Keio University, 2002.

[21] Web Appendix with Erlang code. http://www.dimi.uniud.it/scagnett/pubs/automata-erlang.pdf

Principal Types as Lambda Nets

Pietro Di Gianantonio and Marina Lenisa

Università di Udine, Italy
{pietro.digianantonio, marina.lenisa}@uniud.it

The objective of this work is to present the connections existing among *principal type schemata*, *cut-free* λ*-nets*, and *normal forms* of λ-calculus. As a consequence of these correspondences there exist correspondences among the normalisation algorithms in the above structures, that is: *unification* of principal type schemata, *cut-elimination* on λ-nets, *normalisation* on λ-terms. While the connection between the two latter is well-known, the relationships between unification of principal type schemata and cut-elimination on λ-nets have rarely been presented explicitly. The only works we are aware of are [8, 12], but the presentation is quite different from ours. However, we think that it is worthwhile to analyse in detail such a connection, since it allows to derive a number of properties on the type system from properties of λ-nets. In general, the correspondence existing among types, principal types, λ-nets, and normal forms can be expressed by the following chain of equivalences. For any closed λ-term M:
– M has type τ in the type assignment system, $\vdash M : \tau$, *if and only if*
– M has principal type schema τ' in the principal type assignment system, $\Vdash M : \tau'$, and τ is an instance of τ', *if and only if*
– the λ-net associated to M reduces, by a cut-elimination strategy C, to a cut-free λ-net t, which is the translation of τ', *if and only if*
– M reduces according to the strategy S to a normal form N, and t is the λ-net associated to N.

Once the above correspondences have been established, it is possible to derive properties concerning the type system and the reduction strategy S from properties of λ-nets and the cut-elimination strategy C, or vice-versa.

In this work, we present in detail an instance of the general pattern outlined above. Namely, we choose a particular version of type assignment system, we define a corresponding type assignment system for principal types, and we consider a corresponding notion of λ-net.

We focus on standard λ-calculus, however we present it in terms of its translation into the *linear* $\lambda!$*-calculus*, because this turns out to be convenient in discussing the connection to λ-nets. The type system that we consider includes an intersection operation $\wedge$ on types, which is commutative and associative, but non-idempotent. However, most of our results hold also in the case of idempotency. The type system is almost standard, the only non-standard feature is a bang operator used to mark the type subterms on which the $\wedge$ operator can be applied. Then we introduce principal types (type schemata) together with a type assignment system assigning type schemata to λ-terms. Type schemata differ from simple types by the introduction of type variables and by a box operator, used to mark parts of the type that can be replicated. From type schemata, by suitable instantiation, one gets (ground) types. Each λ-term turns out to have at most one principal type schema, while the set of ground types assignable to a term consists of all the instances of the principal type schema. In the type assignment system for principal types, the rule for introducing the type schema of an application makes use of a generalised *unification algorithm* on types.

We will show that principal types can be seen as an alternative representation of cut-free λ-nets, and the unification algorithm on principal types can be seen as a reformulation of the

cut-elimination algorithm for λ-nets. In order to formalise these correspondences, we present a translation from principal type schemata to λ-nets. However, we need to consider a variation of the standard cut-elimination procedure on λ-nets, whereby the cut-rule concerning the conclusions of a weakening and a promotion does not eliminate the box, but leaves a pending term. With this variation the cut-elimination procedure becomes *perpetual*, that is a cut is eliminated only if the λ-net is strongly normalising. On the other hand, we present, in our setting, the standard translation of λ-terms into λ-nets [9, 1]. The main result of this work can be seen as a commutation result, namely: given a λ-term M, we assign to it a principal type τ, this in turn induces a cut-free λ-net, which can be alternatively obtained by applying to M the standard transformation into λ-nets and then the cut-elimination procedure.

As consequences of the correspondences that we have established, we derive the following results on the type assignment system: the *typable terms* are exactly the *strongly normalising* ones, *subject reduction* holds up-to a suitable relation on types, the *inhabitation* problem is decidable.

The present work builds on [4, 3, 5], where the analogies *involutions as principal types* and *application as unification* have been introduced and explored for a type assignment system related to Abramsky's Geometry of Interaction model of partial involutions.

Related work. Duquesne has first described the connection between principal types and proof-nets, [8]. This approach has been further extended by Regnier in his PhD thesis [12]. Similarly to Duquesne and Regnier approach (referred to as *DR approach* in the following), we use indexes to delimit the parts of a type that need to be replicated. However, there exists a long list of differences. Principal types in DR approach are pairs composed by a variable and a set of pairs of terms that need to be unified; the most general unifier (MGU) of this set of pairs, if existing, will transform the variable in a principal type in our sense. As a result, principal types in DR approach are almost a direct translation of proof-nets, while our principal types describe the normal forms of the proof-nets associated to λ-terms. In defining the principal type of an application, we explicitly define a notion of MGU among two types, while the MGU notion is not present in DR. As far as this aspect, we are in the tradition of Hindley-Milner algorithm for principal types. In DR approach, any lambda-term has a principal type, and the type system characterises weakly normalisable terms, while our system just the strongly normalising ones.

Our work is also related to [10, 11]. Although at first sight the two presentations are quite different, *e.g.* they use a different syntax, and different auxiliary procedures, our type assignment system for principal types is quite similar to the one presented in [10]. What we call *index variable*, there is called *expansion variable*, what we call duplication, there is called expansion. However, the implementation of duplication expansion is described in a completely different form, using different auxiliary structure. A connection between the principal types in [10] and proof-nets is then presented in [11], where it has been used to argue that the type inference with no-idempotent types has the same complexity of normalization. Comparing this work to ours, we can say that our principal type algorithm and the connection between types and proof-nets are simpler and more direct. Moreover, compared to [11], we consider a completely different set of possible applications of the above-mentioned correspondence.

There exists an extensive literature on principal types in general, or more specifically in combination with intersection types, as in our case, [2, 7, 13, 15]. The objective of these works is quite different from ours, namely, often principal types are used to define type inference algorithms for simple programming languages; in other works, the connection between principal types and β-normal forms of terms is investigated [6, 14]. Moreover, the type syntax and the type inference algorithm are quite different from ours, and there is no explicit connection to proof-nets.

References

[1] B. Accattoli. Proof nets and the linear substitution calculus. In *International Colloquium on Theoretical Aspects of Computing*, pages 37–61. Springer, 2018.

[2] A. Bucciarelli, D. Kesner, and D. Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017.

[3] A. Ciaffaglione, P. Di Gianantonio, F. Honsell, M. Lenisa, and I. Scagnetto. Reversible computation and principal types in λ!-calculus. *The Bulletin of Symbolic Logic*, 25(2):931–940, 2019.

[4] A. Ciaffaglione, F. Honsell, M. Lenisa, and I. Scagnetto. The involutions-as-principal types/application-as-unification analogy. In G. Barthe, G. Sutcliffe, and M. Veanes, editors, *LPAR*, volume 57 of *EPiC Series in Computing*, pages 254–270. EasyChair, 2018.

[5] A. Ciaffaglione, F. Honsell, M. Lenisa, and I. Scagnetto. Lambda!-calculus, intersection types, and involutions. In H. Geuvers, editor, *FSCD 2019*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 536–560. Academic Press, 1980.

[7] S. Dolan and A. Mycroft. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, pages 60–72, January 2017.

[8] E. Duquesne and J. Van De Wiele. A new intrinsic characterization of the principal type schemes. Research Report RR-2416, INRIA, 1995. Projet PARA. URL: https://hal.inria.fr/inria-00074259.

[9] S. Guerrini. Proof nets and the lambda-calculus. In T. Ehrhard, editor, *Linear Logic in Computer Science*, pages 316–65. Cambridge University Press, 2004.

[10] A.J. Kfoury and J.B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1):1 – 70, 2004. URL: http://www.sciencedirect.com/science/article/pii/S0304397503005772, doi:https://doi.org/10.1016/j.tcs.2003.10.032.

[11] Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: Why nonlinearity and amnesia make a type system work. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, page 138–149, New York, NY, USA, 2004. Association for Computing Machinery. URL: https://doi.org/10.1145/1016850.1016871, doi:10.1145/1016850.1016871.

[12] L. Regnier. *Lambda-Calcul et Rèseaux*. PhD thesis, Université Paris VII, 1992.

[13] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci.*, 59:181–209, 1988.

[14] Emilie Sayag and Michel Mauny. Characterization of the principal type of normal forms in an intersection type system. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 335–346, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[15] J. B. Wells. The essence of principal typings. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, page 913–925, Berlin, Heidelberg, 2002. Springer-Verlag.

13 Finitary representation of ideal and infinite objects

Coinductive Types from Hangers-and-Pegs Streams

Venanzio Capretta[1]

University of Nottingham, UK
venanzio@duplavis.com

Our purpose is to present a new way to generate elements of coinductive types (sets of tree-like objects with infinitely descending structure) by general stream processes. Our final goal is to set up a flexible methodology to define such processes, offering a more flexible way to generate coinductive objects than the traditional syntactic productivity checks. There are parallels with advanced methods to ensure computational complexity in algorithmics, specifically the technique of amortized analysis.

Our main result is that every coinductive type can be realized as a type of streams with effects. A previous article [7] introduced the notion of *monadic stream*: an infinite sequence of elements, each one triggering a monadic action. This action could spawn several parallel streams (for example, if we use the list monad), so we can use it to represent branching processes and non-well-founded trees: All forms of structured unfolding can be described by a monad, so every coinductive type can be represented by monadic streams.

Inductive and coinductive types can be realized as fixed point of container functors [1, 2]:

Def. 1. *A* container *is a pair* $\langle S, P\rangle$ *consisting of a set of* shapes $S : \mathsf{Set}$ *and a family of* positions $P : S \to \mathsf{Set}$. *It specifies a functor* $(S \rhd P) : \mathsf{Set} \to \mathsf{Set}$*:*

$$(S \rhd P)\, X = \Sigma s : S.P\, s \to X.$$

The initial algebra $\mu(S \rhd P)$ and final coalgebra $\nu(S \rhd P)$ model general inductive and coinductive types. Intuitively their elements are obtained by (non-)well-founded iteration of the functor: trees with shapes as nodes and positions as branching points.

Our work dovetails with research on the construction of a coinductive type via completions of free algebras [6, 4, 3] (for a thorough introduction see the forthcoming book [5]). We can approximate elements of the coinductive type with dependent lists of *slices*, each slice adding a layer of shapes at a certain level [12, 13]. This results in a new container $\langle S^\natural, P^\natural\rangle$, defined by simultaneous induction-recursion [10, 11], which Ghani et al. used to study continuous functions on coinductive types. We used it to obtain a representation theorem for contractions [8]. In both cases, the generation of an element of the coinductive type requires to add a new slice at every stage, that is, we must produce all shapes at a certain level at once.

A more flexible paradigm should permit growing different branches at different rates, allowing more general recursive definitions (and avoiding induction-recursion).

Def. 2. *For a container* $\langle S, P\rangle$*, the container* $\langle S^\triangle, P^\triangle\rangle$ *of* hangers *and* pegs *is defined by:*

$$\begin{array}{lll}
\textbf{data} & S^\triangle : \mathsf{Set} & P^\triangle : S^\triangle \to \mathsf{Set} \\
& \bullet^\triangle : S^\triangle & P^\triangle\, \bullet^\triangle = \mathbb{1} \\
& (\overset{\triangle}{;}) : (s : S)(P\, s \to S^\triangle) \to S^\triangle & P^\triangle\, (s \overset{\triangle}{;} \phi) = \Sigma p : P\, s.P^\triangle\, (\phi\, p)
\end{array}$$

The difference between $\langle S^\natural, P^\natural\rangle$ and $\langle S^\triangle, P^\triangle\rangle$ is that $S^\natural$ is constructed *top-down*, with each slice adding one layer of depth, while $S^\triangle$ is constructed *bottom-up*, with each constructor adding a shape above sub-hangers. The final coalgebra of $(S^\natural \rhd P^\natural)$ is isomorphic to the initial algebra, while the final coalgebra of $(S^\triangle \rhd P^\triangle)$ contains a subset isomorphic to $\nu(S \rhd P)$ (those elements that never use the base constructor $\bullet^\triangle$).

Thm. 1. *The functor $(S^{\triangle} \rhd P^{\triangle})$ is (naturally isomorphic to) the free monad on $(S \rhd P)$, with the following return and bind operations:*

$\mathsf{return} : A \to (S^{\triangle} \rhd P^{\triangle})\, A \quad (\gg\!=) : (S^{\triangle} \rhd P^{\triangle})\, A \to (A \to (S^{\triangle} \rhd P^{\triangle})\, B) \to (S^{\triangle} \rhd P^{\triangle})\, B$
$\mathsf{return}\, a = \langle \bullet^{\triangle}, \lambda \bullet .a \rangle \qquad \langle h, \overrightarrow{a} \rangle \gg\!= g$ *by induction on* h

The definition of the bind operator is by induction on the hanger h:

- *If $h = \bullet^{\triangle}$, then we define $(\langle h, \overrightarrow{a} \rangle \gg\!= g) = g\,(\overrightarrow{a}\,\bullet)$;*
- *If $h = s \overset{\triangle}{;} \phi$ with $s : S$ and $\phi : P\, s \to S^{\triangle}$, we have that $P^{\triangle}\, h = \Sigma p : P\, s . P^{\triangle}\,(\phi\, p)$. So, for every position $p : P\, s$ and every peg $q : P^{\triangle}\,(\phi\, q)$ we have an entry $\overrightarrow{a}_{\langle p,q \rangle} : A$. Using the notation $\overrightarrow{a}_p$ for the function $\lambda q. \overrightarrow{a}_{\langle p,q \rangle}$, we can assume by induction hypothesis on $(\phi\, p)$ that $\langle (\phi\, p), \overrightarrow{a}_p \rangle \gg\!= g$ is defined, with components $\langle k_p, \overrightarrow{b_p} \rangle$. We can now define:*

$$(\langle h, \overrightarrow{a} \rangle \gg\!= g) = \langle (s \overset{\triangle}{;} \lambda p. k_p), (\lambda \langle p, q \rangle . (\overrightarrow{b_p})_q) \rangle.$$

We embed coinductive types into types of *monadic streams* [7]. To every monad we can associate types of streams that trigger actions in it.

Def. 3. *For a monad M and any type A, the type $\mathbb{S}_{M,A}$ of M-streams of A is defined by:*

$$\begin{array}{ll} \textbf{codata} & \mathbb{S}_{M,A} : \mathsf{Set} \\ & \mathsf{mcons}_M : M(A \times \mathbb{S}_{M,A}) \to \mathbb{S}_{M,A} \end{array}$$

If we take M to be the hangers-and-pegs monad, we obtain streams of actions that generate as an effect an approximation to an element of $\nu(S \rhd P)$ (with extra A-labels). These approximations need not converge: A stream could reiterate the empty action $\bullet^{\triangle}$ indefinitely.

The actual entries of the stream (the elements of type A) can be seen as labels on the pegs. We may use them to control the productivity of the stream in a way similar to how *amortized analysis* is used to control the complexity of algorithms [9, Ch.17].

Def. 4. *For $m : \mathbb{N}$, $h : S^{\triangle}$, and $f_1 : P^{\triangle}\, h \to \mathbb{N}$, we define the relation $m \succ f_1$ by cases on h: if $h = \bullet^{\triangle}$ then $(m \succ f_1) = (m > (f_1\, \bullet))$; otherwise $m \succ f_1$ is true.*

We coinductively define a productivity relation between natural numbers and monadic streams:

$$\begin{array}{lll} \textbf{codata} & \mathsf{Productive} : \mathbb{N} \to \mathbb{S}_{(S^{\triangle} \rhd P^{\triangle}),\mathbb{N}} \to \mathsf{Prop} \\ & \mathsf{productive} : & (m : \mathbb{N})(h : S^{\triangle})(f : P^{\triangle}\, h \to \mathbb{N} \times \mathbb{S}_{(S^{\triangle} \rhd P^{\triangle}),\mathbb{N}}) \\ & & m \succ f_1 \wedge \forall q : P^{\triangle}\, h . \mathsf{Productive}\,(f_1\, q)\,(f_2\, q) \\ & & \quad \to \mathsf{Productive}\, m\,(\mathsf{mcons}\,\langle h, f \rangle). \end{array}$$

Thm. 2. *If a stream $\sigma : \mathbb{S}_{(S^{\triangle} \rhd P^{\triangle}),\mathbb{N}}$ satisfies $(\mathsf{Productive}\, m\, \sigma)$ for some number $m : \mathbb{N}$, then there is a unique $\overset{\infty}{\sigma} : \nu(S \rhd P)$ such that the product of the monadic actions of every prefix of σ is an approximation of $\overset{\infty}{\sigma}$.*

The intuitive explanation of this result is that we assign to every peg in the monadic action a numeric value. The next action will generate new pegs: if the action is the empty hanger $\bullet^{\triangle}$, then the value must decrease. There cannot be infinite descending sequences of values, so there cannot be infinite sequences of $\bullet^{\triangle}$ actions. For actions that generate at least one shape, we allow the values to increase arbitrarily. This ensures productivity.

The relation $\succ$ only ensures productivity, but gives no guarantee on the rate of production. If we use more refined relations, we can impose a stricter control on productivity. For example if we require that the new pegs increase the value by adding their depth to it, we will obtain linearly productive streams.

References

[1] Michael Gordon Abbott. *Categories of containers*. PhD thesis, University of Leicester, England, UK, 2003.

[2] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[3] Samson Abramsky. A cook's tour of the finitary non-well-founded sets. In Sergei N. Artëmov, Howard Barringer, Artur S. d'Avila Garcez, Luís C. Lamb, and John Woods, editors, *We Will Show Them! Essays in Honour of Dov Gabbay, Volume One*, pages 1–18. College Publications, 2005.

[4] Jirí Adámek. On final coalgebras of continuous functors. *Theor. Comput. Sci.*, 294(1/2):3–29, 2003.

[5] Jiří Adámek, Stefan Milius, and Lawrence S. Moss. Initial algebras, terminal coalgebras, and the theory of fixed points of functors. Draft book, available from http://www.stefan-milius.eu, 2020.

[6] Michael Barr. Algebraically compact functors. *Journal of Pure and Applied Algebra*, 82:211–231, October 1992.

[7] Venanzio Capretta and Jonathan Fowler. The continuity of monadic stream functions. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[8] Venanzio Capretta, Graham Hutton, and Mauro Jaskelioff. Contractive functions on infinite data structures. In Tom Schrijvers, editor, *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*, pages 5:1–5:13. ACM, 2016.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[10] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[11] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Proceedings of TLCA 1999*, volume 1581 of *LNCS*, pages 129–146. Springer-Verlag, 1999.

[12] Neil Ghani, Peter Hancock, and Dirk Pattinson. Continuous functions on final coalgebras. *Electr. Notes Theor. Comput. Sci.*, 164(1):141–155, 2006. Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS 2006).

[13] Neil Ghani, Peter Hancock, and Dirk Pattinson. Continuous functions on final coalgebras. *Electr. Notes Theor. Comput. Sci.*, 249:3–18, 2009. Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009).

Non-deterministic Functions as Non-deterministic Processes*

Joseph Paulus[1], Daniele Nantes-Sobrinho[2], and Jorge A. Pérez[1]

[1] University of Groningen, The Netherlands [2]University of Brasilia, Brazil

We are interested in rigorously connecting sequential and concurrent programming models via translations of the λ-calculus into the π-calculus. Specifically, we aim at clarifying how properties such as *non-determinism*, *confluence*, and *failure* can be accurately carried over from the sequential realm to the concurrent one, exploiting types. To this end, we consider:

- The *resource calculus* (denoted λ^{R}), a confluent, non-deterministic functional calculus that incorporates failure ("deadlock") [1].
- The *session π-calculus* (denoted $\mathsf{s}\pi$), which supports non-determinism and failure, and follows from the Curry-Howard correspondence connecting linear logic and session types [5].

Our main contribution is a translation from terms in λ^{R} into processes in $\mathsf{s}\pi$, denoted $[\![-]\!]_u$. In the following, we briefly describe the main ingredients and outcomes of our ongoing study.

Context There is a rich history on encodings of the λ-calculus into the π-calculus, triggered by Milner's seminal work [9]. The calculus λ^{R}, first described by Boudol [1], is more expressive than the λ-calculus: it is non-deterministic and allows for confluent reductions. While in the usual λ-calculus an application $M\,N$ considers the term N as a resource infinitely available for substitution, in λ^{R} the term N is considered as a possibly limited resource, extracted from a *bag* of terms. The size of the bag then provides an upper bound to the number of possible substitutions: in [3], a substitution where a resource fails to be provided is considered as a deadlock. Also, the semantics in [3] enforces lazy evaluation and *collapsing* non-determinism (i.e., $M+N$ reduces to either M or N). The semantics in [10, 6] is non-lazy with non-collapsing non-determinism; deadlocks can arise from lack or excess of (linear) resources in the bag.

This Work: Source and Target Calculi For uniformity, we consider sequential and concurrent calculi with confluence and non-collapsing non-determinism. The syntax of λ^{R} is:

$$\begin{array}{lrcl} \text{(Terms)} & M,N,L & ::= & x \mid \lambda x.M \mid (M\ P) \mid M\langle\!\langle P/x\rangle\!\rangle \\ \text{(Bags)} & P,Q,R & ::= & \mathtt{1} \mid \wr M \wr \mid P \cdot Q \\ \text{(Expressions)} & \mathbb{M},\mathbb{N},\mathbb{L} & ::= & M \mid \mathbb{M}+\mathbb{N} \end{array}$$

Terms are unary expressions: they can be variables, abstractions, and applications. Given a bag P and a variable x, $\langle\!\langle P/x\rangle\!\rangle$ denotes an explicit substitution. We write $P \cdot Q$ to denote the concatenation of bags P and Q; this is a commutative and associative operation, where $\mathtt{1}$ is the identity element. Sums are associative and commutative; terms in a sum are silently reordered.

In our semantics for λ^{R}, reductions are lazy (as in [1, 3]) as the explicit substitution of the bag is left delayed and not evaluated; also, non-determinism is non-collapsing (as in [10, 6]). A failure would occur when there is an explicit substitution that can not be performed, i.e., when there are not enough elements for substitution or when there are too many elements in the bag.

In our semantics, the λ^{R} term $M_0 = (\lambda x.x\wr x\wr)\wr M\wr \cdot \wr N\wr$ reduces as follows:

$$(\lambda x.x\wr x\wr)\wr M\wr \cdot \wr N\wr \longrightarrow x\wr x\wr\langle\!\langle \wr M\wr \cdot \wr N\wr /x\rangle\!\rangle \longrightarrow M\wr x\wr\langle\!\langle \wr N\wr /x\rangle\!\rangle + N\wr x\wr\langle\!\langle \wr M\wr /x\rangle\!\rangle$$

*Paulus and Pérez are partially supported by the Netherlands Organization for Scientific Research (NWO) under the VIDI Project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

First, the redex becomes an explicit substitution of the bag $\wr M \wr \cdot \wr N \wr$; then, all elements in the bag are non-deterministically and linearly substituted into the head variable.

The concurrent calculus $\mathsf{s}\pi$, introduced in [5], is a session π-calculus with (internal) non-determinism and failure, which rests upon a Curry-Howard isomorphism connecting an extended linear logic and session types for disciplining message-passing concurrency. Its syntax is:

$$P, Q ::= \begin{array}{l} \mathbf{0} \mid \overline{x}(y).P \mid x(y).P \mid (P \mid Q) \mid (\nu x)P \mid [x \leftrightarrow y] \mid x.\overline{\texttt{close}} \mid x.\texttt{close}; P \mid \\ P \oplus Q \mid x.\texttt{some}_{\overline{w}}; P \mid x.\overline{\texttt{some}}; P \mid x.\overline{\texttt{none}} \end{array}$$

Constructs in the first line have standard readings: the inactive process, output and input prefixes, parallel composition, restriction, name forwarding, and closing of a session, respectively. Constructs in the second line represent internal non-determinism and potentially abortable session behaviors. $P \oplus Q$ specifies a non-deterministic choice of behaving as P or Q. This is non-collapsing determinism: a contextual rule allows $P \oplus Q \longrightarrow P' \oplus Q$ if $P \longrightarrow P'$. Process $x.\texttt{some}_{\overline{w}}; P$, where $\overline{w} = w_1, \ldots, w_n$ is a sequence of channels dependent on x, requires the confirmation of a session behaviour along channel x in order to perform P. Process $x.\overline{\texttt{some}}; P$ (resp. $x.\overline{\texttt{none}}$) specify the availability (resp. failure) of producing a session along x:

$$x.\overline{\texttt{some}}; P \mid x.\texttt{some}_{\overline{w}}; Q \longrightarrow P \mid Q \qquad\qquad x.\overline{\texttt{none}} \mid x.\texttt{some}_{\overline{w}}; Q \longrightarrow w_1.\overline{\texttt{none}} \mid \ldots \mid w_n.\overline{\texttt{none}}$$

Our Encoding As usual in translations of λ-calculi in the π-calculus, our encoding $[\![-]\!]_u$ is parameterised over a channel u along which the behaviour of the encoded term is provided. The term $M_0 = (\lambda x.x\wr x\wr)\wr M \wr \cdot \wr N \wr$, discussed above, is encoded into $\mathsf{s}\pi$ as follows:

$$(\nu v)\big(R \mid v.\texttt{some}_w; v\langle x\rangle.(x.\texttt{some}_w; x\langle x_1\rangle.(x_1.\texttt{some}_{\overline{w}}; [\![M]\!]_{x_1} \mid x\langle x_2\rangle.(x_2.\texttt{some}_{\overline{w}}; [\![N]\!]_{x_2} \mid x.\overline{\texttt{close}})) \mid [v \leftrightarrow u])\big)$$
$$\oplus$$
$$(\nu v)\big(R \mid v.\texttt{some}_w; v\langle x\rangle.(x.\texttt{some}_w; x\langle x_1\rangle.(x_1.\texttt{some}_{\overline{w}}; [\![N]\!]_{x_1} \mid x\langle x_2\rangle.(x_2.\texttt{some}_{\overline{w}}; [\![M]\!]_{x_2} \mid x.\overline{\texttt{close}})) \mid [v \leftrightarrow u])\big)$$

where R is defined as:

$$\begin{array}{l} R = v.\overline{\texttt{some}}; v(x).x.\overline{\texttt{some}}; x(x_1).x(x_2).x.\texttt{close}; (\nu t)(x_1.\overline{\texttt{some}}; [x_1 \leftrightarrow t] \mid \\ \quad t.\texttt{some}_w; t\langle y\rangle.(y.\texttt{some}_w; y\langle y_1\rangle.(y_1.\texttt{some}_{\overline{w}}; x_2.\overline{\texttt{some}}; [x_2 \leftrightarrow y_1] \mid y.\overline{\texttt{close}}) \mid [t \leftrightarrow v])) \end{array}$$

Names always first either confirm their behaviour or wait for their behaviour to be confirmed. When a inputs a variable it must confirm its behaviour; when a channel outputs a fresh channel it must first confirm that a name can be received. This is induced by the type discipline in [5].

To simplify the encoding of substitutions in λ^{R} as processes in $\mathsf{s}\pi$, we have introduced an intermediate calculus: the *resource calculus with sharing*, denoted $\lambda^{\mathsf{R}}_{\mathsf{S}}$, which borrows elements from the atomic λ-calculus [8] and the resource control calculus [7]. In particular, in $\lambda^{\mathsf{R}}_{\mathsf{S}}$ variables can occur at most once; multiple occurrences can be specified with the sharing operator $[\widetilde{x} \leftarrow x]$.

We have developed type systems for λ^{R} and $\lambda^{\mathsf{R}}_{\mathsf{S}}$ based on intersection types. This is a natural choice, as known type systems for the resource control calculus [7] and the resource calculi in [2, 10] rely on intersection types. We adapt the type system in [4] to better match the behavior of the typed $\mathsf{s}\pi$ processes. The main difference with respect to [2, 10] is that all terms in a bag must be of the same type. Also, our type system rules out terms that will produce failures.

Our encoding of λ^{R} into $\mathsf{s}\pi$ (mediated by $\lambda^{\mathsf{R}}_{\mathsf{S}}$) preserves types (i.e., well-typed terms are encoded into well-typed processes) and is operational correspondent. We have also developed another encoding, which concerns *well-formed* λ^{R} terms as source calculus. The class of well-formed processes contains well-typed processes but includes also terms that may fail. By transforming potentially untyped λ^{R} terms into well-typed $\mathsf{s}\pi$ processes, our second encoding gives a justification of the concept of failure as typed concurrent processes.

References

[1] G. Boudol. The lambda-calculus with multiplicities (abstract). In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, pages 1–6, 1993.

[2] G. Boudol, P. Curien, and C. Lavatelli. A semantics for lambda calculi with resources. *Mathematical Structures in Computer Science*, 9(4):437–482, 1999.

[3] G. Boudol and C. Laneve. Lambda-calculus, multiplicities, and the pi-calculus. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 659–690, 2000.

[4] A. Bucciarelli, T. Ehrhard, and G. Manzonetto. Categorical models for simply typed resource calculi. *Electr. Notes Theor. Comput. Sci.*, 265:213–230, 2010.

[5] L. Caires and J. A. Pérez. Linearity, control effects, and behavioral types. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 229–259, 2017.

[6] M. Dominici, S. Ronchi Della Rocca, and P. Tranquilli. Standardization in resource lambda-calculus. In *Proceedings 2nd International Workshop on Linearity, LINEARITY 2012, Tallinn, Estonia, 1 April 2012.*, pages 1–11, 2012.

[7] S. Ghilezan, J. Ivetic, P. Lescanne, and S. Likavec. Intersection types for the resource control lambda calculi. In *Theoretical Aspects of Computing - ICTAC 2011 - 8th International Colloquium, Johannesburg, South Africa, August 31 - September 2, 2011. Proceedings*, pages 116–134, 2011.

[8] T. Gundersen, W. Heijltjes, and M. Parigot. Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 311–320, 2013.

[9] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[10] M. Pagani and S. Ronchi Della Rocca. Solvability in resource lambda-calculus. In *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 358–373, 2010.

A Type-Theoretic Potpourri: Towards Final Coalgebras of Accessible Functors

Henning Basold[1] and Niccolò Veltri[2]

[1] LIACS – Leiden University
h.basold@liacs.leidenuniv.nl
[2] Tallinn University of Technology
niccolo@cs.ioc.ee

The semantics of finitely-branching transition systems, process calculi [8] and various kinds of probabilistic processes [6] can be modelled as final coalgebras of accessible functors [10]. This result is folklore in classical set theory and has been used to extend the data types of Isabelle [4], but it was so far not exploitable in constructive type theory. In this talk, we will present a type theory that combines several ideas from recent years to obtain a constructive theory of final coalgebras for accessible functors.

The main ideas that go into this theory are

- set-truncated higher inductive types [1, 3, 9], also called quotient inductive types, to model accessible functors as types with free types variables,
- coinductive types as fixed points of types with a free type variable,
- an equality type that is explicitly axiomatised with reflexivity, function extensionality and a coinduction proof principle,
- a combination of cubical type theory (CTT) and the coercion from observational type theory (OTT), similar to what was suggested by Chapman et al. [5] and further developed by Sterling et al. [7], to handle the extensional equality type and preserve canonicity.

As for the last point, the combination of CTT and OTT, we chose a different route, however, than the mentioned authors. The equality type in our theory is neither defined by induction on types nor do we introduce matching on types. Instead, we push all the work to the reduction relation, which only enables reductions if a coercion is of the right shape. We also do not allow abstraction over interval variables, as is customary in CTT, but explicitly add function extensionality as a term constructor for equality proofs. Interestingly, this necessitates β-reduction rules for extensionality and coinduction proofs. Finally, we avoid for the time being the introduction of universes, large elimination and thereby univalence into our theory. To still be able to formulate dependent elimination of inductive types and the coinduction principle, we lift types with free type variables to predicates and relations by using inductive families, see the first authors thesis [2, Sec. 7.4].

A Glance at the Theory Let us briefly present here the main parts of our type theory. The cubical influence on the theory is use of interval expressions to describe endpoints of equality proofs and coercions. To this end, every judgement is equipped with an interval context Δ and we can form interval expressions from variables in Δ, 0, 1 and affine combinations:

$$\frac{\Delta \vdash p, q, r : \mathbb{I}}{\Delta \vdash p(q - r) : \mathbb{I}} \qquad (p(q - r))(q' - r') \longrightarrow p((q(q' - r')) - (r(q' - r')))$$

We display here the reduction rule for interactions of affine combinations on the right, while the other reduction rules for affine combinations are the obvious ones.

Next, the theory features type equalities, coercions along those equalities and the corresponding reduction rules. We indicate types in reductions whenever they are necessary.

$$\frac{\Delta \mid \Gamma \vdash A, B \ \ \mathbf{Ty}}{\Delta \mid \Gamma \vdash A \sim B \ \ \mathbf{TyEq}} \qquad \frac{\Delta \mid \Gamma \vdash Q : A \sim B \quad \Delta \vdash p : \mathbb{I}}{\Delta \mid \Gamma \vdash Q\ p \ \ \mathbf{Ty}}$$

$$\frac{\Delta, i : \mathbb{I} \mid \Gamma \vdash M \ \ \mathbf{Ty} \quad M[0/i] \equiv A \quad M[1/i] \equiv B}{\Delta \mid \Gamma \vdash \langle i \rangle . M : A \sim B}$$

$$\frac{\Delta \mid \Gamma \vdash Q : A \sim B \quad \Delta \vdash p, q : \mathbb{I} \quad \Delta \mid \Gamma \vdash t : Q\ p}{\Delta \mid \Gamma \vdash t[p \mid Q \mid q\rangle : Q\ q}$$

$$\frac{\Delta \mid \Gamma \vdash Q : A \sim B}{\Delta \mid \Gamma \vdash Q\ 0 \longrightarrow A \ \ \mathbf{Ty}} \qquad \frac{\Delta \mid \Gamma \vdash Q : A \sim B}{\Delta \mid \Gamma \vdash Q\ 1 \longrightarrow B \ \ \mathbf{Ty}} \qquad (\langle i \rangle . M)\ p \longrightarrow M[p/i] \qquad Q \longrightarrow \langle i \rangle . Q\ i$$

$$t[0 \mid Q \mid 0\rangle \longrightarrow t \qquad t[1 \mid Q \mid 1\rangle \longrightarrow t \qquad t[1 \mid Q \mid 0\rangle \longrightarrow t[0 \mid \langle i \rangle . Q\ i(1-0) \mid 1\rangle$$

The last case for the reduction of coercions, namely $t[0 \mid Q \mid 1\rangle$, is defined by case distinction on the type of t, similarly to theory of Chapman et al. [5].

For brevity, we leave out the rules for higher inductive types, which essentially combine set-truncated HIT [3] with a universe-free syntax for families and liftings [2]. Coinductive types are simply fixed point types $\nu X.\, A$ of types with one free variable. The rules of the type theory ensure that the variable X only appears in strictly positive position.

The final ingredient is the axiomatisation of the (heterogeneous) equality type on terms. We only expose here reflexivity and function extensionality here to simplify the presentation. Apart from the standard cubical computation rules for term equalities, we also admit β-reductions for the reflexivity and function extensionality. These are necessary to obtain canonical forms. The coinduction principle is introduced in the same way, but requires lifting of types to relations, which is definable by induction on the types to be lifted, cf. [2, Sec. 7.4].

$$\frac{\Delta \mid \Gamma \vdash s : A \quad \Delta \mid \Gamma \vdash t : B}{\Delta \mid \Gamma \vdash s \sim t \ \ \mathbf{Ty}} \qquad \frac{\Delta \mid \Gamma \vdash s, t : A \quad \Delta \mid \Gamma \vdash Q : s \sim t \quad \Delta \vdash p : \mathbb{I}}{\Delta \mid \Gamma \vdash Q\ p : A}$$

$$\frac{\Delta \mid \Gamma \vdash s : A}{\Delta \mid \Gamma \vdash \mathsf{refl}\ s : s \sim s} \qquad \frac{\Delta \mid \Gamma \vdash f, g : \Pi(x : A).B \quad \Delta \mid \Gamma, x : A \vdash e : f\ x \sim g\ x}{\Delta \mid \Gamma \vdash \mathsf{funext}\ f\ g\ (x.\, e) : f \sim g}$$

$$\frac{\Delta \mid \Gamma \vdash s, t : A \quad \Delta \mid \Gamma \vdash Q : s \sim t}{\Delta \mid \Gamma \vdash Q\ 0 \longrightarrow s : A} \qquad \frac{\Delta \mid \Gamma \vdash s, t : A \quad \Delta \mid \Gamma \vdash Q : s \sim t}{\Delta \mid \Gamma \vdash Q\ 1 \longrightarrow t : A}$$

$$\frac{\Delta \mid \Gamma \vdash s : A \quad \Delta \vdash p : \mathbb{I}}{\mathsf{refl}\ s\ p \longrightarrow s : A}$$

$$\frac{\Delta \mid \Gamma \vdash f, g : \Pi(x : A).B \quad \Delta \mid \Gamma, x : A \vdash e : f\ x \sim g\ x \quad \Delta \vdash p : \mathbb{I} \quad \Delta \mid \Gamma \vdash t : A}{\Delta \mid \Gamma \vdash \mathsf{funext}\ f\ g\ (x.\, e)\ p\ t \longrightarrow e[t/x]\ p : B[t]}$$

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. *Mixed Inductive-Coinductive Reasoning: Types, Programs and Logic*. PhD Thesis, Radboud University, 2018.

[3] 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.

[4] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *Proceedings of ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.

[5] J. Chapman, F. N. Forsberg, and C. McBride. The Box of Delights (Cubical Observational Type Theory). Unpublished Note, Jan. 2018.

[6] A. Sokolova. Probabilistic systems coalgebraically: A survey. *TCS*, 412(38):5095–5110, 2011.

[7] J. Sterling, C. Angiuli, and D. Gratzer. Cubical Syntax for Reflection-Free Extensional Equality. In H. Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] D. Turi and G. D. Plotkin. Towards a Mathematical Operational Semantics. In *LICS'97*, pages 280–291. IEEE, 1997.

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

[10] J. Worrell. Terminal sequences for accessible endofunctors. In *Coalgebraic Methods in Computer Science, CMCS 1999, Amsterdam, The Netherlands, March 20-21, 1999*, pages 24–38, 1999.

Resolving finite indeterminacy
A definitive constructive universal prime ideal theorem

Peter Schuster and Daniel Wessel

Università degli Studi di Verona, Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona, Italy
{peter.schuster,daniel.wessel}@univr.it

Ideal objects and the transfinite methods that grant their existence abound in abstract algebra. From a logical point of view, ideal objects serve for proving the semantic conservation of additional non-deterministic sequents, that is, with finite but not necessarily singleton succedents. By design, dynamical proofs [14, 20, 39] allow to eliminate the use of ideal methods upon shifting focus from semantic model extension principles to their corresponding syntactic conservation theorems. This move in line with Hilbert's programme has shaped modern constructive algebra and has seen tremendous success, not least because coherent (or first-order geometric) logic predominates in that area [10, 14, 22, 25]: the use of a non-deterministic axiom can be captured by a finite branching of the proof tree. Coherent theories, on the other hand, lend themselves to automated theorem proving [3, 4, 15, 35].

A paradigmatic case, which to a certain extent has been neglected in dynamical algebra proper, is Krull's Lemma for prime ideals, presumably the most prominent consequence of the axiom of choice in ring theory. A particular form of this asserts that a multiplicative subset of a commutative ring contains the zero element if and only if the set at hand meets every prime ideal. Prompted by certain aspects in the novel treatment of valuative dimension [18], Krull's Lemma has seen a constructive treatment only recently [33]. The latter, however, has brought us to unearth the underlying general phenomenon in the present paper: Whenever an algebraic or proof certificate is obtained by the semantic conservation of certain additional non-deterministic axioms, there is a finite labelled tree belonging to a suitable inductively generated class which tree encodes the desired computation.

Our characterisation works in the fairly universal setting of *consequence relations*, which serve here to capture basic structures, e.g., ideals of commutative rings, propositional theories, or partial orders, on top of which we consider certain non-deterministic axioms that describe "ideal" specifications of the former: prime ideals, complete theories, or linear order extensions.

Recall that a consequence relation on a set S is a relation $\rhd$ between finite subsets[1] and elements of S, which is *reflexive, monotone* and *transitive* in the following sense:

$$\frac{U \ni a}{U \rhd a}\ (\mathrm{R}) \qquad \frac{U \rhd a}{U, V \rhd a}\ (\mathrm{M}) \qquad \frac{U \rhd b \quad U, b \rhd a}{U \rhd a}\ (\mathrm{T})$$

where the usual shorthand notations are in place.[2] The *ideals* of a consequence relation are the subsets $\mathfrak{a}$ of S such that if $\mathfrak{a} \supseteq U$ and $U \rhd a$, then $a \in \mathfrak{a}$. If U is a finite subset of S, then $\langle U \rangle = \{\, a \in S \mid U \rhd a \,\}$ is an ideal.

A decisive aspect of our approach is the notion of a regular set for certain non-deterministic axioms over a fixed consequence relation, where by a *non-deterministic axiom*[3] on S we under-

[1] We understand a set to be *finite* if it can be written as $\{ a_1, \ldots, a_n \}$ for some $n \geqslant 0$.

[2] The converse relation $\lhd$ corresponds to a finitary formal topology [6, 23, 29].

[3] Our terminology borrows from van den Berg's principle of *non-deterministic inductive definitions* [36], variants of which have recently come to play a role in constructive reverse mathematics [16, 17].

stand a pair (A, B) of finite subsets of S. A subset $\mathfrak{p}$ of S is *closed* for (A, B) if $A \subseteq \mathfrak{p}$ implies $\mathfrak{p} \between B$, where the latter is to say that $\mathfrak{p}$ and B have an element in common.[4]

Let $\mathcal{E}$ be a set of non-deterministic axioms over $\rhd$. A *prime ideal* is an ideal of $\rhd$ that is closed for every element of $\mathcal{E}$. For instance, if $\rhd$ denotes deduction, and $\mathcal{E}$ consists of all pairs $(\emptyset, \{\, \varphi, \neg\varphi \,\})$ for sentences φ, then the (prime) ideals are exactly the (complete) theories.

A subset R of S is *regular* with respect to $\mathcal{E}$ if, for all finite subsets U of S and all $(A, B) \in \mathcal{E}$,

$$\frac{(\forall b \in B)\ \langle U, b\rangle \between R}{\langle U, A\rangle \between R}$$

Abstracted from the multiplicative subsets occurring in Krull's Lemma, regular sets turn out to be the fundamental ingredient of our *Universal Prime Ideal Theorem*:

Proposition 1 (**ZFC**). *A subset R of S is regular if and only if, for every ideal $\mathfrak{a}$, we have $R \between \mathfrak{a}$ precisely when $R \between \mathfrak{p}$ for all prime ideals $\mathfrak{p} \supseteq \mathfrak{a}$.*

Regular sets further account for the constructive version of Proposition 1. To this end, given an ideal $\mathfrak{a}$, we next define a collection $T_{\mathfrak{a}}$ of *finite* labelled trees such that the root of every $t \in T_{\mathfrak{a}}$ be labelled with a finite subset U of $\mathfrak{a}$, and the non-root nodes be labelled with elements of S. The latter will be determined successively by consequences of U along the elements of $\mathcal{E}$.

We understand paths, which necessarily are finite, to lead from the root of a tree to one of its leaves. Given a path π of $t \in T_{\mathfrak{a}}$, we write $\pi \rhd a$ whenever $U, b_1, \ldots, b_n \rhd a$ where U labels the root of t and $b_1, \ldots, b_n$ are the labels occurring at the non-root nodes of π. In any such context, let also $\langle \pi \rangle = \{a \in S : \pi \rhd a\}$.

Definition. Let $\mathfrak{a}$ be an ideal. We generate $T_{\mathfrak{a}}$ inductively according to the following rules:

1. For every finite $U \subseteq \mathfrak{a}$, the trivial tree labelled with U belongs to $T_{\mathfrak{a}}$.
2. If $(A, B) \in \mathcal{E}$ and if $t \in T_{\mathfrak{a}}$ has a path π such that $\pi \rhd a$ for every $a \in A$, then add, for every $b \in B$, a child labelled with b at the leaf of π.

We say that $t \in T_{\mathfrak{a}}$ *terminates* in $R \subseteq S$ if, for every path π of t, there is $r \in R$ such that $\pi \rhd r$.

Here is our *Constructive Universal Prime Ideal Theorem*, which already works in a suitable fragment of Aczel's Constructive Zermelo–Fraenkel set theory **CZF** [1, 2].

Proposition 2 (**CZF**). *A subset R of S is regular if and only if, for every ideal $\mathfrak{a}$, we have $R \between \mathfrak{a}$ precisely when there is a tree $t \in T_{\mathfrak{a}}$ which terminates in R.*

In this manner we manage to uniformise many of the known instances of the dynamical method [14, 20, 39]. We further generalise the universal proof-theoretic conservation criterion that has been offered before [27], using Scott–style entailment relations [34],[5] to unify the several phenomena present in the literature, e.g. [5, 12, 19, 21, 24].

Last but not least, we thus link the syntactical with the semantic approach:

Proposition 3 (**CZF**). *Let $\mathfrak{a}$ be an ideal, and $t \in T_{\mathfrak{a}}$ a tree. For every prime ideal $\mathfrak{p} \supseteq \mathfrak{a}$ there is a path π through t such that $\mathfrak{p} \supseteq \langle \pi \rangle$, that is, $\mathfrak{p}$ contains all the labels occurring along π.*

In other words, every ideal object can be approximated from inside by one of the corresponding tree's branches. We thus abstract from the case of commutative rings we have considered before [33].

[4] We borrow this symbol from formal topology [30, 31].

[5] The relevance of multi-conclusion entailment to constructive algebra and point-free topology has been pointed out in [5], and has been used very widely, e.g. in [7–9, 12, 13, 24, 26, 28, 32, 37, 38]. Lorenzen's precedence is currently under scrutiny [11].

References

[1] Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical report, Institut Mittag–Leffler, 2000. Report No. 40.

[2] Peter Aczel and Michael Rathjen. Constructive set theory. Book draft, 2010. URL: https://www1.maths.leeds.ac.uk/~rathjen/book.pdf.

[3] Marc Bezem and Thierry Coquand. Automating coherent logic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 246–260. Springer, 2005.

[4] Marc Bezem and Dimitri Hendriks. On the mechanization of the proof of Hessenberg's theorem in coherent logic. *J. Automat. Reason.*, 40(1):61–85, 2008.

[5] Jan Cederquist and Thierry Coquand. Entailment relations and distributive lattices. In Samuel R. Buss, Petr Hájek, and Pavel Pudlák, editors, *Logic Colloquium '98. Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic, Prague, Czech Republic, August 9–15, 1998*, volume 13 of *Lect. Notes Logic*, pages 127–139. A. K. Peters, Natick, MA, 2000.

[6] Francesco Ciraulo and Giovanni Sambin. Finitary formal topologies and Stone's representation theorem. *Theoret. Comput. Sci.*, 405(1–2):11–23, 2008.

[7] Thierry Coquand. About Stone's notion of spectrum. *J. Pure Appl. Algebra*, 197(1–3):141–158, 2005.

[8] Thierry Coquand. Space of valuations. *Ann. Pure Appl. Logic*, 157:97–109, 2009.

[9] Thierry Coquand and Henri Lombardi. Hidden constructions in abstract algebra: Krull dimension of distributive lattices and commutative rings. In M. Fontana, S.-E. Kabbaj, and S. Wiegand, editors, *Commutative Ring Theory and Applications*, volume 231 of *Lect. Notes Pure Appl. Mathematics*, pages 477–499, Reading, MA, 2002. Addison-Wesley.

[10] Thierry Coquand and Henri Lombardi. A logical approach to abstract algebra. *Math. Structures Comput. Sci.*, 16:885–900, 2006.

[11] Thierry Coquand, Henri Lombardi, and Stefan Neuwirth. Lattice-ordered groups generated by an ordered group and regular systems of ideals. *Rocky Mountain J. Math.*, 49(5):1449–1489, 2019.

[12] Thierry Coquand and Henrik Persson. Valuations and Dedekind's Prague theorem. *J. Pure Appl. Algebra*, 155(2–3):121–129, 2001.

[13] Thierry Coquand and Guo-Qiang Zhang. Sequents, frames, and completeness. In Peter G. Clote and Helmut Schwichtenberg, editors, *Computer Science Logic (Fischbachau, 2000)*, volume 1862 of *Lecture Notes in Comput. Sci.*, pages 277–291. Springer, Berlin, 2000.

[14] Michel Coste, Henri Lombardi, and Marie-Françoise Roy. Dynamical method in algebra: Effective Nullstellensätze. *Ann. Pure Appl. Logic*, 111(3):203–256, 2001.

[15] John Fisher and Marc Bezem. Skolem machines. *Fund. Inform.*, 91(1):79–103, 2009.

[16] Ayana Hirata, Hajime Ishihara, Tatsuji Kawai, and Takako Nemoto. Equivalents of the finitary non-deterministic inductive definitions. *Ann. Pure Appl. Logic*, 170(10):1256–1272, 2019.

[17] Hajime Ishihara and Takako Nemoto. Non-deterministic inductive definitions and fullness. In D. Probst and P. Schuster, editors, *Concepts of Proof in Mathematics, Philosophy, and Computer Science*, volume 6 of *Ontos Mathematical Logic*, pages 163–170. Walter de Gruyter, Berlin, 2016.

[18] Gregor Kemper and Ihsen Yengui. Valuative dimension and monomial orders. 2019. URL: https://arxiv.org/abs/1906.12067.

[19] Henri Lombardi. Hidden constructions in abstract algebra. I. Integral dependance. *J. Pure Appl. Algebra*, 167:259–267, 2002.

[20] Henri Lombardi and Claude Quitté. *Commutative Algebra: Constructive Methods. Finite Projective Modules*, volume 20 of *Algebra and Applications*. Springer Netherlands, Dordrecht, 2015.

[21] Christopher J. Mulvey and Joan Wick-Pelletier. A globalization of the Hahn–Banach theorem. *Adv. Math.*, 89:1–59, 1991.

[22] Sara Negri. Geometric rules in infinitary logic. In *Arnon Avron on Semantics and Proof Theory*

of Non-Classical Logics, Outstanding Contributions to Logic. Springer. Forthcoming.

[23] Sara Negri. Stone bases alias the constructive content of Stone representation. In Aldo Ursini and Paolo Aglianò, editors, *Logic and algebra. Proceedings of the international conference dedicated to the memory of Roberto Magari, April 26–30, 1994, Pontignano, Italy*, volume 180 of *Lecture Notes in Pure and Applied Mathematics*, pages 617–636. Marcel Dekker, New York, 1996.

[24] Sara Negri, Jan von Plato, and Thierry Coquand. Proof-theoretical analysis of order relations. *Arch. Math. Logic*, 43:297–309, 2004.

[25] Michael Rathjen. Remarks on Barr's theorem. Proofs in geometric theories. In D. Probst and P. Schuster, editors, *Concepts of Proof in Mathematics, Philosophy, and Computer Science*, volume 6 of *Ontos Mathematical Logic*, pages 347–374. Walter de Gruyter, Berlin, 2016.

[26] Davide Rinaldi. *Formal Methods in the Theories of Rings and Domains*. Doctoral dissertation, Universität München, 2014.

[27] Davide Rinaldi, Peter Schuster, and Daniel Wessel. Eliminating disjunctions by disjunction elimination. *Indag. Math. (N.S.)*, 29(1):226–259, 2018. Communicated first in *Bull. Symb. Logic* 23 (2017), 181–200.

[28] Davide Rinaldi and Daniel Wessel. Extension by conservation. Sikorski's theorem. *Log. Methods Comput. Sci.*, 14(4:8):1–17, 2018.

[29] Giovanni Sambin. Intuitionistic formal spaces—a first communication. In D. Skordev, editor, *Mathematical Logic and its Applications, Proc. Adv. Internat. Summer School Conf., Druzhba, Bulgaria, 1986*, pages 187–204. Plenum, New York, 1987.

[30] Giovanni Sambin. Some points in formal topology. *Theoret. Comput. Sci.*, 305(1–3):347–408, 2003.

[31] Giovanni Sambin. *The Basic Picture. Structures for Constructive Topology*. Oxford Logic Guides. Clarendon Press, Oxford, forthcoming.

[32] Konstantin Schlagbauer, Peter Schuster, and Daniel Wessel. Der Satz von Hahn–Banach per Disjunktionselimination. *Confluentes Math.*, 11(1):79–93, 2019.

[33] Peter Schuster, Daniel Wessel, and Ihsen Yengui. Dynamic evaluation of integrity and the computational content of Krull's lemma. 2019. Preprint.

[34] Dana Scott. Completeness and axiomatizability in many-valued logic. In Leon Henkin, John Addison, C.C. Chang, William Craig, Dana Scott, and Robert Vaught, editors, *Proceedings of the Tarski Symposium (Proc. Sympos. Pure Math., Vol. XXV, Univ. California, Berkeley, Calif., 1971)*, pages 411–435. Amer. Math. Soc., Providence, RI, 1974.

[35] Sana Stojanović, Vesna Pavlović, and Predrag Janičić. A coherent logic based geometry theorem prover capable of producing formal and readable proofs. In *International Workshop on Automated Deduction in Geometry*, pages 201–220. Springer, 2010.

[36] Benno van den Berg. Non-deterministic inductive definitions. *Arch. Math. Logic*, 52(1–2):113–135, 2013.

[37] Daniel Wessel. Ordering groups constructively. *Comm. Algebra*, 47(12):4853–4873, 2019.

[38] Daniel Wessel. Point-free spectra of linear spreads. In S. Centrone, S. Negri, D. Sarikaya, and P. Schuster, editors, *Mathesis Universalis, Computability and Proof*, Synthese Library, pages 353–374. Springer, 2019.

[39] Ihsen Yengui. *Constructive Commutative Algebra. Projective Modules over Polynomial Rings and Dynamical Gröbner Bases*, volume 2138 of *Lecture Notes in Mathematics*. Springer, Cham, 2015.

14 Terms, rewriting and types

Confluence in Higher-Order Theories

Gaspard Férey and Jean-Pierre Jouannaud

INRIA, LSV, ENS Paris-Saclay, Université Paris-Saclay, France

Introduction Extending conversion with user-defined higher-order rewrite rules is becoming a standard in proof assistants based on intuitionistic type theory [2, 4]. In dependent type theories however, confluence is often needed to prove that rewriting preserves key properties of β-reduction such as type preservation, strong normalization, consistency and decidability of type checking.

In a series of papers, we develop techniques based on van Oostroms decreasing diagrams [6], that reduce confluence proofs to the checking of various forms of critical pairs for higher-order rewrite rules extending β-reduction on pure lambda-terms. These results can be applied to various encodings of type theories in Dedukti [4, 1], assuming they preserve types.

Higher-Order Rewriting Functional reductions on pure λ-terms can be enriched with higher-order term rewriting. We use Klop's framework which distinguishes a set of variables used for abstractions, from a set of meta-variables with arities used for the higher-order rules. Substituting a meta-variable requires enough abstractions in order to perform the associated beta-reductions *on site* [5]. A term U with meta-variables $\mathcal{MV}ar(U)$ is called a meta-term.

Definition 1 (Higher-Order Rewriting)**.** *A rewrite rule $L \longrightarrow R$ is a pair of meta-terms such that L is a pattern and $\mathcal{MV}ar(R) \subseteq \mathcal{MV}ar(L)$. Then $u \xrightarrow[L\to R]{p} v$ iff $u|_p = L\gamma$ and $v = u[R\gamma]_p$.*

Higher-order rewriting doesn't behave well with substitution: whenever $\sigma \longrightarrow \gamma$, then $t\sigma \longrightarrow\!\!\!\!\!\rightarrow t\gamma$, complicating the confluence proofs with β. It can however be extended by grouping several steps together, yielding parallel reductions (at *parallel* positions), orthogonal reductions (at *overlap-free* positions), and *sub-rewriting*: $\xrightarrow{p} \subset \Longrightarrow^{P} \subset \otimes\!\!\Longrightarrow^{O} \subset \underset{\equiv}{\longrightarrow} \subset \underset{\equiv}{\Longrightarrow} \subset \longrightarrow\!\!\!\!\!\rightarrow$

$$\frac{\sigma \longrightarrow \gamma}{t\sigma \Longrightarrow t\gamma} \qquad \frac{t \Longrightarrow u \qquad \sigma \otimes\!\!\Longrightarrow \gamma}{t\sigma \otimes\!\!\Longrightarrow u\gamma} \qquad \frac{t \longrightarrow u \qquad \forall X \in \mathcal{MV}ar(t)\, \forall i\, (X^i\sigma \equiv X\gamma)}{\bar{t}\sigma \underset{\equiv}{\longrightarrow} u\gamma}$$

where $\bar{t}$ is a linearized copy of t obtained by renaming the i occurrence of meta-variable X into X^i.

Left-Linear Theories If $\mathcal{R}$ is a set of left-linear rewrite rules, we have $\underset{\beta}{\Longleftarrow\!\!\otimes} \underset{\mathcal{R}}{\longrightarrow} \subseteq \underset{\mathcal{R}}{\Longrightarrow} \underset{\beta}{\Longleftarrow\!\!\otimes}$ which is a decreasing diagram if we chose a label for β bigger than the labels for $\mathcal{R}$.

This implies however that critical peaks may not rely on arbitrary β steps to be joined (only those included in the substitutions for meta-variables used for higher-order rewriting survive).

Theorem 1 (Confluence)**.** *Let $\mathcal{R}$ be a left-linear rewrite system. The relation $\underset{\mathcal{R}}{\longrightarrow} \cup \underset{\beta}{\longrightarrow}$ is confluent in the following three cases:*

- *$\mathcal{R}$ is terminating and locally confluent [3]*
- *labels of R-rewrites are smaller than those of β-rewrites, right-hand sides of rules have no embedded meta-variables, and the parallel critical pairs of R have decreasing diagrams.*
- *labels of R-rewrites are biggest, and orthogonal critical pairs have a decreasing diagram.*

In the latter two cases, R-rewrite steps use rule-labeling: the label of a step is the number of the rule which is used (parallel and orthogonal steps must use a *single rule* at each step).

Non-Left-Linear Theories The case of non-left-linear rules is counter-intuitive: Klop showed that the confluence of β-reductions is not preserved by the rule $\mathbf{F}(X,X) \to X$. The counter examples rely on fixpoints operators applied to instances of a non-linear left-hand side. To achieve confluence in presence of such rules, one needs to restrict the set of terms considered. For example, the rule above preserves the confluence of simply typed rules.

The idea is therefore to come up with a simple type system that will eliminate the counter-examples, but accept, say, dependently typed terms. The system we present below, which uses natural number as base types, accepts all pure λ-terms. However, when given an environment defining constants, not all simply typable terms are accepted yet. We illustrate the idea with the rule $F(X,X) \to X$. Variables and occurrences of F are decorated as in x^n and $\mathbf{F}_n$. Meta-variables are not decorated. The typing system is as follows:

$$\frac{}{x^n : n} \qquad \frac{}{\mathbf{F}_n : n \to n \to n+1} \qquad \frac{\mathbf{s} \in \mathcal{S}}{\mathbf{s} : 0} \qquad \frac{t : n \qquad u : k}{(t\ u) : \max(n,k)} \qquad \frac{t : k \leq n}{\lambda x^n.t : n}$$

$$\frac{t : n \to A \qquad u : k \leq n}{(t\ u) : A} \qquad \frac{t : \overline{A} \to k}{\lambda x^n.t : n \to \overline{A} \to \max(k,n)}$$

Note that any term that does not contain the symbol $\mathbf{F}$, such as the infamous $(\lambda x.(x\ x)\ \ \lambda x.(x\ x))$, can be typed with 0 using the first five rules if we choose to decorate all variables with 0. Hence, all pure λ-terms are accepted by the typing system. Occurences of applications of $\mathbf{F}$ lift the type of their arguments up, so that, for an example, $\lambda z.(\lambda x.(\mathbf{F}\ x\ z)\ \lambda y.(\mathbf{F}\ y\ z)) : 0 \to 2$ if we chose the decoration z^0, y^0, x^1.

In fact whenever $t : n$ is derivable, then n is a bound by the maximal depth of nested $\mathbf{F}$ symbols in subterms of t *and their reducts*. As a consequence, the type of a term is non-increasing by taking subterms and β-reductions and F-reductions. An important consequence is that occurrences of $\mathbf{F}$ in a typable term $t : n$ are fully applied, and decorated by some $k < n$. So, $t = t[(\mathbf{F}_k\ u\ v)]_p : n$, and $u : l < n$ and $v : l' < n$ (l, l', not k, because of subtyping).

Moving to the confluence proof, we can label β (resp. F) -reductions with the decoration n of the corresponding abstracted variable (resp. $\mathbf{F}$ symbol), hence defining $\xrightarrow[\beta]{n}$ and $\xrightarrow[F]{n}$. We then apply van Oostrom's decreasing diagram technique to the rewrite relations $\overset{n}{\underset{\beta}{\otimes\!\!\Longrightarrow}}$, and $\overset{n}{\underset{\equiv_n}{\Longrightarrow}}$, where $\equiv_n$ is the congruence defined by all reductions of label smaller than n strictly. The label of a step, as just defined, will serve as its label in van Oostrom's decreasing diagram technique.

The above properties guarantee then that reductions below non-linear meta-variables in F-redices have a label strictly less than that of the F reduction above. This provides the following decreasing diagrams:

$$\overset{n}{\underset{\equiv_n}{\Longleftarrow}}\overset{k<n}{\underset{\equiv_k}{\Longrightarrow}} \subseteq \overset{k<n}{\underset{\equiv_k}{\Longleftarrow}}\overset{n}{\underset{\equiv_n}{\Longrightarrow}} \qquad \overset{n}{\underset{\equiv_n}{\Longleftarrow}}\overset{n}{\underset{\equiv_n}{\Longrightarrow}} \subseteq \overset{n}{\underset{\equiv_n}{\Longleftarrow}}\overset{n}{\underset{\equiv_n}{\Longrightarrow}} \qquad \overset{n}{\underset{\beta}{\Longleftarrow\!\!\otimes}}\overset{n}{\underset{\beta}{\otimes\!\!\Longrightarrow}} \subseteq \overset{n}{\underset{\beta}{\otimes\!\!\Longrightarrow}}\overset{n}{\underset{\beta}{\Longleftarrow\!\!\otimes}}$$

Theorem 2. *Rewriting with the non-left-linear rule* $\mathbf{F}\ X\ X \longrightarrow \mathbf{a}$ *preserves the confluence of* β *on typable decorated terms.*

The theorem can be generalized to a higher-order rewrite system $\mathcal{R}$, even in presence of critical pairs, provided the type of the right-hand side of rules is at most equal to the type of their left-hand side, and the (typable) critical pairs have decreasing diagrams.

The question which has not been resolved yet is whether confluence in our type system implies confluence for simply typed terms, or for dependently typed terms, both are important for applications. In general, the answer is likely to be negative, and it will then be necessary to modify the type system to get more typable terms. Our approach, we believe, is promising.

References

[1] Ali Assaf, Guillaume Burel, Raphal Cauderlier, Gilles Dowek, Catherine Dubois, Frdric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a Logical Framework based on the lambda-pi-Calculus Modulo Theory. draft, INRIA, 2019.

[2] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. How to tame your rewrite rules, 2018. Draft.

[3] Gaspard Ferey and Jean-Pierre Jouannaud. Confluence in (un)typed higher-order type theories i. draft. hal-, INRIA, march 2019. available from http://dedukti.gforge.inria.fr/.

[4] Gilles Dowek at all. The Dedukti system, 2016. Available from http://dedukti.gforge.inria.fr/.

[5] J.W. Klop. *Combinatory Reduction Systems.* Number 127 in Mathematical Centre Tracts. CWI, Amsterdam, The Netherlands, 1980. PhD Thesis.

[6] Vincent van Oostrom. Confluence by decreasing diagrams. *Theor. Comput. Sci.*, 126(2):259–280, 1994.

Type safety of rewriting rules in dependent types

Frédéric Blanqui

Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria
Laboratoire Spécification et Vérification, 94235, Cachan, France

Abstract

The expressiveness of dependent type theory can be extended by identifying types modulo some additional computation rules [BFG97, Bla05, CA]. But, for preserving the decidability of type-checking or the logical consistency of the system, one must make sure that those user-defined rewriting rules preserve typing. In this talk, I will present a new method to check that property using Knuth-Bendix completion. A prototype implementation for the Dedukti proof assistant is available on https://github.com/wujuihsuan2016/lambdapi/tree/sr.

We consider the simplest dependent type theory $\lambda\Pi$ but identify types equivalent modulo β-reduction and some set $\mathcal{R}$ of user-defined rewriting rules.

The relation $\to_{\mathcal{R}}$ generated by $\mathcal{R}$ preserves typing if, for all rule $l \to r \in \mathcal{R}$, environments Γ, substitutions σ and terms A, $\Gamma \vdash r\sigma : A$ whenever $\Gamma \vdash l\sigma : A$.

A first idea is to require:

(*) there exist Δ and B such that $\Delta \vdash l : B$ and $\Delta \vdash r : B$,

but this condition is not sufficient in general:

Example 1. Consider the rule $f(xy) \to y$ with $f : B \Rightarrow B$. In the environment $\Delta = x : B \Rightarrow B, y : B$, we have $\Delta \vdash l : B$ and $\Delta \vdash r : B$. However, in the environment $\Gamma = x : A \Rightarrow B, y : A$, we have $\Gamma \vdash l : B$ and $\Gamma \vdash r : A$.

The condition (*) is sufficient for proving subject-reduction when the rule left-hand side is a non-variable simply-typed first-order term [BFG97, Proposition 3.13], a notion that can be easily extended to $\lambda\Pi$.

However, (*) is not satisfactory in the context of dependent types. Indeed, when function symbols have dependent types, it often happens that a term is typable only if it is non-linear. And, with non-left-linear rewriting rules, $\to = \to_\beta \cup \to_{\mathcal{R}}$ is generally not confluent on untyped terms [Klo80], while there exist many confluence criteria for left-linear rewriting systems [vO94].

Example 2. Consider the following rule to define the *tail* function on vectors:

$$\mathit{tail}\ n\ (\mathit{cons}\ x\ p\ v) \to v$$

where $\mathit{tail} : \forall n : N, V(sn) \Rightarrow Vn$, $V : N \Rightarrow \star$, $\mathit{nil} : V0$, $\mathit{cons} : \alpha \Rightarrow \forall n : N, Vn \Rightarrow V(sn)$, $\alpha : \star$, $0 : N$ and $s : N \Rightarrow N$. For the left-hand side to be typable, we need $p = n$ because $\mathit{tail}\ n$ expects an argument of type $V(sn)$ but $\mathit{cons}\ x\ p\ v$ is of type $V(sp)$. Yet, the rule preserves typing. Indeed, assume that there is an environment Γ, a substitution σ and a term A such that $\Gamma \vdash \mathit{tail}\ n\sigma\ (\mathit{cons}\ x\sigma\ p\sigma\ v\sigma) : A$. By inversion of typing rules, we get $V(n\sigma) \simeq A$, $\Gamma \vdash A : s$ for some sort s, $V(sp\sigma) \simeq V(sn\sigma)$ and $\Gamma \vdash v\sigma : Vp\sigma$, where $\simeq$ is the smallest congruence containing $\to$. Assume now that V and s are undefined, that is, there is no rule of $\mathcal{R}$ of the form $Vt \to u$ or $st \to u$. Then, by confluence, $n\sigma \simeq p\sigma$. Therefore, $Vp\sigma \simeq A$ and $\Gamma \vdash v\sigma : A$.

Hence, that a rewriting rule $l \to r$ preserves typing does not mean that its left-hand side l must be typable. Actually, if no instance of l is typable, then $l \to r$ trivially preserves typing (since it can never be applied)! The point is therefore to check that any typable instance of $l \to r$ preserves typing [Bla05].

The new algorithm that we propose for checking that a rule $l \to r$ preserves typing proceeds in three steps. First, we generate conversion constraints that are satisfied by every typable instance of l, by taking a fresh variable $\widehat{y}$ for the type of a variable y:

$$\frac{}{y \uparrow \widehat{y}\,[\emptyset]} \qquad \frac{f : \forall x_1 : T_1, \ldots, \forall x_n : T_n, U \quad t_1 \uparrow A_1[\mathcal{E}_1] \quad t_n \uparrow A_n[\mathcal{E}_n]}{ft_1 \ldots t_n \uparrow U\sigma[\mathcal{E}_1 \cup \ldots \cup \mathcal{E}_n \cup \{A_1 = T_1\sigma, \ldots, A_n = T_n\sigma\}] \text{ where } \sigma = \{(x_1, t_1), \ldots, (x_n, t_n)\}}$$

Example 3. We have $cons\ x\ p\ v \uparrow V(sp)[\mathcal{E}_1]$ where $\mathcal{E}_1 = \{\widehat{x} = \alpha, \widehat{p} = N, \widehat{v} = Vp\}$, and $tail\ n\ (cons\ x\ p\ v) \uparrow Vn[\mathcal{E}_2]$ where $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$. This means that, if σ is a substitution and $(tail\ n\ (cons\ x\ p\ v))\sigma$ is typable, then $V(sp\sigma) \simeq V(sn\sigma)$.

Second, we simplify the conversion constraints by taking into account the injectivity of the product and other function symbols:

$$\begin{array}{rcl} \mathcal{D} \uplus \{t = u\} & \rightsquigarrow & \mathcal{D} \cup \{t' = u'\} \text{ if } t \to^* t' \text{ and } u \to^* u' \\ \mathcal{D} \uplus \{\forall x : t_1, t_2 = \forall x : u_1, u_2\} & \rightsquigarrow & \mathcal{D} \cup \{t_1 = u_1, t_2 = u_2\} \text{ if } x \text{ is fresh} \\ \mathcal{D} \uplus \{ft_1 \ldots t_n = fu_1 \ldots u_n\} & \rightsquigarrow & \mathcal{D} \cup \{t_i = u_i \mid i \in I\} \\ & & \text{if } f \text{ is } I\text{-injective and } \forall i \notin I, t_i \simeq u_i \end{array}$$

Example 4. In the previous example, we can replace the constraint $V(sp) = V(sn)$ by $p = n$ if V and s are not defined by any rewriting rules.

Finally, we check whether r has the same type as l in the type system where the conversion relation is extended with the equational theory generated by the constraints inferred in the first step. For this system to be decidable and implementable using Dedukti itself, we use Knuth-Bendix completion [KB70] to replace the set of conversion constraints by an equivalent but convergent (*i.e.* terminating and confluent) set of rewriting rules, which always terminates on closed equations when one uses a total order on function symbols [BN98].

Example 5. By taking $\widehat{x} > \widehat{v} > \widehat{p} > \widehat{n} > V > \alpha > N > s > p > n$ (but any other total order would work as well), Knuth-Bendix completion yields the rewriting system $\mathcal{D} = \{\widehat{x} \to \alpha, \widehat{p} \to N, \widehat{v} \to Vn, \widehat{n} \to N, p \to n\}$. It is then possible to prove that $\to_\beta \cup \to_\mathcal{R} \cup \to_\mathcal{D}$ is convergent if $\to_\beta \cup \to_\mathcal{R}$ is convergent, $\mathcal{R}$ is left-linear and V and s are undefined. Finally, we are left to prove that the type of v, $\widehat{v}$, is equivalent to Vn, which is the case since Vn is the $\to_\mathcal{D}$-normal form of $\widehat{v}$.

References

[BN98] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[BFG97] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.

[Bla05] F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

[CA] J. Cockx and A. Abel. Sprinkles of Extensionality for Your Vanilla Type Theory (abstract). Presented at TYPES'2016.

[KB70] D. Knuth and P. Bendix. Simple word problems in universal algebra. In *Computational problems in abstract algebra,* , p. 263–297. Pergamon Press, 1970.

[Klo80] J. W. Klop. *Combinatory reduction systems.* PhD thesis, Utrecht Universiteit, NL, 1980. Published as Mathematical Center Tract 129.

[vO94] V. van Oostrom. *Confluence for abstract and higher-order rewriting.* PhD thesis, Vrije Universiteit Amsterdam, NL, 1994.