23rd International Conference on Types for Proofs and Programs

TYPES 2017

Budapest, Hungary, 29 May - 1 June 2017

Abstracts

Department of Programming Languages and Compilers,
Faculty of Informatics, Eötvös Loránd University

Budapest, 2017

23rd International Conference on Types for Proofs and Programs,
TYPES 2017
Budapest, Hungary, 29 May - 1 June 2017
Abstracts
http://types2017.elte.hu

Edited by Ambrus Kaposi

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
Pázmány Péter sétány 1/C, 1117 Budapest, Hungary
http://www.inf.elte.hu

Cover design by András Fórizs

ISBN 978-963-284-883-9 (print)
ISBN 978-963-284-884-6 (pdf)

Preface

This volume contains the abstracts of the talks presented at the *23rd International Conference on Types for Proofs and Programs, TYPES 2017*, to take place in Budapest, Hungary, 29 May - 1 June 2017.

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

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

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 A formal post-proceedings volume is prepared after the conference; papers submitted to that must represent unpublished work and are subjected to a full review process.

The programme of TYPES 2017 includes three invited talks by Edwin Brady (University of St Andrews), Sara Negri (Unviersity of Helsinki) and Jakob Rehof (TU Dortmund). The contributed part of the programme consists of 50 talks.

Similarly to the 2011 and the 2013-2016 editions of the conference, the post-proceedings of TYPES 2015 will appear in Dagstuhl's *Leibniz International Proceedings in Informatics (LIPIcs)* series.

We are grateful for the support of COST Action CA15123 EUTypes.

May 17, 2017 Ambrus Kaposi and Tamás Kozsik
Budapest

Organisation

Program Committee

Andreas Abel	Gothenburg University
Thorsten Altenkirch	University of Nottingham
José Carlos Espírito Santo	University of Minho
Fredrik Forsberg	University of Strathclyde
Silvia Ghilezan	University of Novi Sad
Hugo Herbelin	INRIA,France
Martin Hofmann	LMU Munich
Ambrus Kaposi	Eötvös Loránd University (co-chair)
Tamás Kozsik	Eötvös Loránd University (co-chair)
Assia Mahboubi	INRIA, France
Alexandre Miquel	University of the Republic, Uruguay
Keiko Nakata	SAP Potsdam
Andrew Polonsky	University of Bergen
Simona Ronchi Della Rocca	University of Torino
Aleksy Schubert	University of Warsaw
Wouter Swierstra	Utrecht University
Tarmo Uustalu	Tallinn University of Technology

Organising committee

Ambrus Kaposi, Tamás Kozsik, András Kovács and the Department of Programming Languages and Compilers

Host

Faculty of Informatics at Eötvös Loránd University

Sponsor

COST Action CA15123 EUTypes

cost
EUROPEAN COOPERATION
IN SCIENCE & TECHNOLOGY

Table of Contents

An Architecture for Dependently Typed Applications in Idris

Edwin Brady

University of St Andrews
ecb10@st-andrews.ac.uk

A useful pattern in dependently typed programming is to define a state transition system, for example the states and operations in a network protocol, as an indexed monad. We index each operation by its input and output states, thus guaranteeing that operations satisfy pre- and post-conditions, by typechecking. However, what if we want to write a program using several systems at once? What if we want to define a high level state transition system, such as a network application protocol, in terms of lower level states, such as network sockets and mutable variables?

In this talk, I will present an architecture for dependently typed applications in the Idris programming language, based on a hierarchy of state transition systems, and implemented in a generic data type ST. This is based on a monad indexed by contexts of resources, allowing us to reason about multiple state transition systems in the type of a function. Using ST, I will show: how to implement a state transition system as a dependent type, with type level guarantees on its operations; how to account for operations which could fail; how to combine state transition systems into a larger system; and, how to implement larger systems as a hierarchy of state transition systems. I will illustrate the system with a high level network application protocol, implemented in terms of POSIX network sockets.

Proof systems based on neighbourhood semantics

Sara Negri

University of Helsinki
sara.negri@helsinki.fi

A method is presented for generating proof systems on the basis of neighbourhood semantics. The method is a generalisation of the internalisation of possible worlds semantics into sequent calculi. Its specific features will be illustrated through case studies from the treatment of counterfactuals and conditional beliefs.

Bounding Principles for Decision Problems with Intersection Types

Jakob Rehof

TU Dortmund
jakob.rehof@cs.tu-dortmund.de

The classical type-theoretic decision problems, inhabitation (given a type, is there a term having the type?) and typability (given a term, does it have a type?), are undecidable for intersection types. But one can consider bounding principles admitting of algorithmic solutions while retaining interesting levels of expressive power. We overview the status of known bounding principles, computational aspects of the associated decision problems, and some application perspectives.

Normalization by Evaluation for Sized Dependent Types

Andreas Abel[1], Andrea Vezzosi[1], and Théo Winterhalter[2]

[1] Department of Computer Science and Eng., Gothenburg University, Sweden
{abela,vezzosi}@chalmers.se
[2] École Normale Supérieure de Cachan, France
theo.winterhalter@ens-cachan.fr

Abstract

Sized types have been developed to make termination checking more perspicuous, more powerful, and more modular by integrating termination into type checking. In dependently-typed proof assistants where proofs by induction are just recursive functional programs, the termination checker is an integral component of the trusted core, as validity of proofs depends on termination. However, a rigorous integration of full-fledged sized types into dependent type theory is lacking so far. Such an integration is non-trivial, as explicit sizes in proof terms might get in the way of equality checking, making terms appear distinct that should have the same semantics. In this work, we integrate dependent types and sized types with higher-rank size polymorphism, which is essential for generic programming and abstraction. We introduce a size quantifier ∀ which lets us ignore sizes in terms for equality checking, alongside with a second quantifier Π for abstracting over sizes that do affect the semantics of types and terms. Judgmental equality is decided by an adaptation of normalization-by-evaluation.

Agda [5] features first-class size polymorphism [1] in contrast to theoretical accounts of sized dependent types [6, 7, 10] who typically just have prenex (ML-style) size quantification. Consequently, Agda's internal language contains size expressions in terms wherever a size quantifier is instantiated. However, these size expressions, which are not unique due to subtyping, can get in the way of reasoning about sizeful programs. Consider the type of sized natural numbers.

```
data Nat : Size → Set where
  zero : ∀ i → Nat (i + 1)
  suc  : ∀ i → Nat i → Nat (i + 1)
```

We define subtraction $x \dot{-} y$ on natural numbers, sometimes called the monus function, which computes $\max(0, x - y)$. It is defined by induction on the size j of the second argument y, while the output is bounded by size i of the first argument x. (The input-output relation of monus is needed for a natural implementation of Euclidean divison.)

```
monus : ∀ i → Nat i → ∀ j → Nat j → Nat i
monus i         x         .(j + 1) (zero j)  = x
monus .(i + 1) (zero i)   .(j + 1) (suc j y) = zero i
monus .(i + 1) (suc i x)  .(j + 1) (suc j y) = monus i x j y
```

We wish to prove that subtracting x from itself yields 0, by induction on x. The case $x = 0$ should be trivial, as $x \dot{-} 0 = x$ by definition, hence, $0 \dot{-} 0 = 0$. A simple proof by reflexivity should suffice. However, the goal shows a mismatch between size ∞ and size i coming from the computation of monus $(i + 1)$ (zero i) $(i + 1)$ (zero i).

```
monus-diag : ∀ i → (x : Nat i) → zero ∞ ≡ monus i x i x
monus-diag .(i + 1) (zero i)  = {! zero ∞ ≡ zero i !} -- goal
monus-diag .(i + 1) (suc i x) = monus-diag i x
```

The proof could be completed by an application of reflexivity if Agda ignored sizes where they act as *type argument*, i. e., in constructors and term-level function applications, but not in types where they act as regular argument, e. g., in $\mathsf{Nat}\,i$.

The problem is solved by distinguishing relevant (Π) from irrelevant ($\forall$) size quantification. The relevant quantifier is the usual dependent function space over sizes. In particular, the congruence rule for size application requires matching size arguments:

$$\frac{\Gamma \vdash t = t' : \Pi i.\, T \qquad \Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash t\, a = t'\, a : T[a/i]}$$

Typically, the relevant quantifier is used in types of types, for instance, in its non-dependent form, in $\mathsf{Nat} : \mathsf{Size} \to \mathsf{Set}$. In contrast, the irrelevant size quantifier is used in types of programs and ignores sizes in size applications. The rules for application, while Church-style, *de facto* implement Curry-style quantification:

$$\frac{\Gamma \vdash t = t' : \forall i.\, T \qquad \Gamma^{\oplus} \vdash a, a', b : \mathsf{Size}}{\Gamma \vdash t\, a = t'\, a' : T[b/i]} \qquad \frac{\Gamma \vdash t : \forall i.\, T \qquad \Gamma^{\oplus} \vdash a, b : \mathsf{Size}}{\Gamma \vdash t\, a : T[b/i]}$$

Further, the size arguments are scoped in the *resurrected* [9] context $\Gamma^{\oplus}$ and, thus, are allowed to mention irrelevant size variables. Those are introduced by irrelevant size abstraction and marked by the $\div$-symbol in the context. In contrast, the quantified size variable may occur relevantly in the *type*.

$$\frac{\Gamma, i \div \mathsf{Size} \vdash t : T}{\Gamma \vdash \lambda i.\, t : \forall i.\, T} \qquad \frac{\Gamma, i{:}\mathsf{Size} \vdash T : \mathsf{Set}}{\Gamma \vdash \forall i.\, T : \mathsf{Set}}$$

The lack of type unicity arising from the size application rule has prevented us from adopting the usual incremental algorithm for equality checking [8, 3]. However, we have succeeded to employ normalization by evaluation [2] for deciding judgmental equality [4].

References

[1] Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In *FICS 2012*, volume 77 of *EPTCS*, pages 1–11, 2012. Invited talk.

[2] Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. In *LICS'07*, pages 3–12. IEEE CS Press, 2007.

[3] Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *LMCS*, 8(1:29):1–36, 2012. TYPES'10 special issue.

[4] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. Normalization by evaluation for sized dependent types. Accepted for presentation at ICFP, 2017.

[5] AgdaTeam. The Agda Wiki, 2017.

[6] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. CIC^: Type-based termination of recursive definitions in the Calculus of Inductive Constructions. In *LPAR'06*, volume 4246 of *LNCS*, pages 257–271. Springer, 2006.

[7] Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *RTA'04*, volume 3091 of *LNCS*, pages 24–39. Springer, 2004.

[8] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM TOCL*, 6(1):61–101, 2005.

[9] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS'01*, pages 221–230. IEEE CS Press, 2001.

[10] Jorge Luis Sacchini. Type-based productivity of stream definitions in the calculus of constructions. In *LICS'13*, pages 233–242. IEEE CS Press, 2013.

A Type Theory with Native Homotopy Universes

Robin Adams[1] and Andrew Polonsky[2]

[1] Universitetet i Bergen
[2] University Paris Diderot

We present a type theory called $\lambda{\simeq}_2$ with an extensional equality relation. The universe of types is closed under the logical relation defined by induction on the structure of types.

The type system has three universes:

- The trivial universe **1**, which has one object $*$.
- The universe **Prop** of *propositions*. An object of **Prop** is called a *proposition*, and the objects of a proposition are called *proofs*.
- The universe **Set** of *sets*.
- The universe **Groupoid** of *groupoids*.

The system has been designed in such a way that it should be straightforward to extend the system with three, four, . . . dimensions.

For each universe U, we have an associated relation of equality between U-types $\simeq$, and between objects of U-equal types $\sim$. The associated rules of deduction are:

$$\frac{A : U \quad B : U}{A \simeq B : U} \qquad \frac{a : A \quad e : A \simeq B \quad b : B}{a \sim_e b : U^-}$$

where U^- is the universe one dimension below U. Thus:

- Given two propositions ϕ and ψ, we have the proposition $\phi \simeq \psi$ which denotes the proposition 'ϕ if and only if ψ'. If $\delta : \phi$, $\epsilon : \psi$ and $\chi : \phi \simeq \psi$, then $\delta \sim_\chi \epsilon = \mathbf{1}$. (*Cf* In homotopy type theory, any two objects of a proposition are equal.)
- Given two sets A and B, we have the set $A \simeq B$, which denotes the set of all bijections between A and B. Given $a : A$, $f : A \simeq B$ and $b : B$, we have the proposition $a \sim_f b$: **Prop**, which denotes intuitively that a is mapped to b by the bijection f.
- Given two groupoids G and H, we have the groupoid $G \simeq H$, which denotes the groupoid of all groupoid equivalences between G and H. Given $g : G$, $\phi : G \simeq H$ and $h : H$, we have the set $g \sim_\phi h$: **Set**, which can be thought of as the set of all paths between $\phi(g)$ and h in H.

The introduction and elimination rules for $\simeq$ ensure that $A \simeq B$ is the type of equivalences between A and B.

$$\frac{A : U}{1_A : A \simeq A} \qquad \frac{a : A}{r_a : a \sim_{1_A} a} \qquad \frac{e : A \simeq B \quad a : A}{e^+(a) : B} \qquad \frac{e : A \simeq B \quad b : B}{e^-(b) : A}$$

$$\frac{a : A \quad b : B \quad e : A \simeq B}{e^=(a,b) : (a \sim_{1_A} e^-(b)) \simeq (e^+(a) \sim_{1_B} b)} \qquad \frac{\begin{array}{rl}\Gamma, x : A & \vdash b : B \\ \Gamma, y : B & \vdash a : A \\ \Gamma, x : A, y : B & \vdash e : (x \sim_{1_A} a) \simeq (b \sim_{1_B} y)\end{array}}{\Gamma \vdash \mathsf{univ}([x : A]b, [y : B]a, [x : A, y : B]e) : A \simeq B}$$

Each universe is itself an object of the next universe; thus $\mathbf{1} : \mathbf{Prop} : \mathbf{Set} : \mathbf{Groupoid}$. We also have the following definitional equalities: $\phi \sim_{1_{\mathbf{Prop}}} \psi \overset{\text{def}}{=} \phi \simeq \psi$, $A \sim_{1_{\mathbf{Set}}} B \overset{\text{def}}{=} A \simeq B$.

As well as the normal operation of substitution, we have an operation of *path substitution*:

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash e : a \sim_{1_A} a'}{\Gamma \vdash b[x//e] : b[x/a] \sim_{B[x//e]} b[x/a']}$$

The system $\lambda{\simeq_2}$ enjoys the following properties. Univalence holds definitionally; that is, an equality between types $A \simeq B$ is exactly (definitionally) the type of equivalences between A and B. Also, transport respects reflexivity and composition definitionally.

This type theory has been formalised in Agda, using the method of the system `Kipling` from McBride [5]. The method ensures that, if s and t are definitionally equal expressions in $\lambda \simeq_2$, then $[\![s]\!]$ and $[\![t]\!]$ are definitionally equal objects in Agda. We interpret each context with a groupoid in Agda; that is, we define the following type of contexts and functions:

```
data Cx : Set₁

[_]C : Cx → Set₁

EQC : ∀ Γ → [ Γ ]C → [ Γ ]C → Set
EQC₂ : ∀ {Γ} {a₁ a₂ b₁ b₂ : [ Γ ]C} →
   EQC Γ a₁ a₂ → EQC Γ b₁ b₂ → EQC Γ a₁ b₁ → EQC Γ a₂ b₂ → Set
```

The formalisation is available online at `https://github.com/radams78/Equality2`.

Related Work An earlier version of this system was presented in [2]. In this talk, we also give semantics to this system in Agda's type theory extended with a native type of groupoids, and show how the syntax and semantics are formalised in Agda.

Our system is closely related to the system PHOML (Predicative Higher-Order Minimal Logic) presented in [1]. The system $\lambda{\simeq_2}$ can be seen as an extension of PHOML with groupoids, and with a univalent equality for both sets and groupoids.

Cubical type theory [3, 4] has a very similar motivation to this work, and also offers a type theory with univalence and a computational interpretation. One difference with our system is that in cubical type theory, transport respects reflexivity and composition only up to propositional equality.

References

[1] Robin Adams, Marc Bezem, and Thierry Coquand. A normalizing computation rule for propositional extensionality in higher-order minimal logic. Submitted for publication in TYPES 2016. `https://arxiv.org/abs/1610.00026`, 2017.

[2] Robin Adams and Andrew Polonsky. A type system with native homotopy universes. Talk given at Workshop on Homotopy Type Theory / Univalent Foundations, Porto, Portugal, 2016. `http://hott-uf.gforge.inria.fr/andrew.pdf`.

[3] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[4] Simon Huber. Canonicity for cubical type theory. `arXiv:1607.04156`, 2016.

[5] Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In Bruno C. d. S. Oliveira and Marcin Zalewksi, editors, *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP 2010)*, pages 1–12. ACM, 2010.

Monads from multi-sorted binding signatures

Benedikt Ahrens[1], Ralph Matthes[2], and Anders Mörtberg[3]

[1] Inria Rennes Bretagne Atlantique, Nantes, France
[2] Institut de Recherche en Informatique de Toulouse (IRIT), CNRS and Université Paul Sabatier
[3] Inria Sophia Antipolis – Méditerranée, Sophia Antipolis, France

We present a construction in UniMath of simply-typed term systems with variable binding. Specifically, we start out with a very simple notion of 'multi-sorted binding signature'. Such a signature abstractly specifies a simply-typed language with variable binding. To any such signature S we construct a monad, say, T_S that deserves to be called the 'syntax generated by the signature S'. This monad associates, to any context Γ, the well-typed terms $T_S(\Gamma)$ in that context. The monadic multiplication represents a well-behaved substitution operation on the terms of T_S. We illustrate what this means in the examples below.

We are not the first to work on the topic of constructing (some notion of) syntax from signatures; similar constructions have been carried out in [6, 3], for instance, both working in the proof assistant Coq. The fundamental difference to those previous efforts is that we work in a spartan core type theory—UniMath—that does not have general inductive types. Indeed, one of the main challenges we have met is the construction of some class of inductive families of sets from the type constructors available in UniMath. This construction is topos-theoretic in spirit, akin to the constructions carried out in [7, 9].

For sake of simplicity, we illustrate first the simpler case of *untyped* syntax. In this case, a binding signature is given by a family of lists of natural numbers. Each member of the family specifies a constructor, and the associated list of natural numbers specifies, via its length, the number of arguments of that constructor, and the number of variables bound in each argument. For instance, the untyped lambda calculus LC is specified by the signature $\{\mathsf{abs} := [1]\ ,\ \mathsf{app} := [0,0]\}$ specifying the constructors $\mathsf{abs}_X : \mathsf{LC}(\mathsf{option}(X)) \to \mathsf{LC}(X)$ and $\mathsf{app}_X : \mathsf{LC}(X) \times \mathsf{LC}(X) \to \mathsf{LC}(X)$. The functor $\mathsf{LC} : \mathsf{Set} \to \mathsf{Set}$ thus obtained is actually a monad on the category of sets, and, more specifically, the 'initial monad' in a suitable category [8]. Note that the constructor for variables is handled separately, and is not specified in the binding signature.

In previous work [2], we have constructed, in UniMath, the 'term monad' specified by any binding signature. In the present work, we generalize the notion of signature to allow for the specification of multi-sorted term systems, over a set of sorts, and generalize the construction of the term monad to those multi-sorted binding signatures.

The multi-sorted binding signatures we consider are similar to those of [6, 3]. They specify a set sort of sorts and typing and binding behaviour of the constructors. For instance, the signature of the simply-typed lambda calculus requires a set of sorts equipped with a binary function type constructor $(\Rightarrow)$. The signature is then given by two families, both indexed over two sorts s, t, with 'arities' $\big\langle [\langle [\,], s \Rightarrow t\rangle, \langle [\,], s\rangle], t\big\rangle$ and $\big\langle [\langle [s], t\rangle], s \Rightarrow t\big\rangle$, corresponding to application and λ-abstraction, respectively. To such a multi-sorted binding signature over a set sort of sorts we associate a monad on the slice category $\mathsf{Set}/\mathsf{sort}$.

We now outline the construction of a monad from a multi-sorted binding signature. The construction proceeds in several steps, and uses results provided in previous work.

Signature with strength from a multi-sorted binding signature. A 'signature with strength' is a more general, and more semantic notion of signature. It consists of an endofunctor H on endofunctors (on a suitable base category) together with extra data θ that specifies information on 'how to do substitution' on H-algebras. The first step consists of associating a suitable signature with strength (H, θ) on $\mathsf{Set}/\mathsf{sort}$ to any binding signature S over sort.

Initial algebra for H_S. The next step consists in constructing an initial algebra for H, or, more precisely, for $\underline{\mathsf{Id}} + H$, the sum of H with the functor constantly the identity on the base category. The summand $\underline{\mathsf{Id}}$ encodes 'variables as terms', whereas the functor H specifies the constructors given via S. Due to the absence of general inductive types in UniMath, this step requires the formalization of some category-theoretic machinery in UniMath, and it constitutes the main technical contribution of the present work.

Substitution on the initial algebra. The initial algebra constructed in the previous step consists, in particular, of an endofunctor T on $\mathsf{Set}/\mathsf{sort}$, and a natural transformation $\eta : \mathsf{Id} \to T$. We complement the pair (T, η) to a monad by constructing a suitable 'substitution' operation. This step can itself be divided into two steps: first, we construct a substitution system [10] from (T, η), employing 'Generalized Mendler Iteration'. This iteration scheme is explained in [5] and formalized in [2]. In a second step, we apply a map from substitution systems to monads on the substitution systems thus obtained. This map has been constructed in [10] and been extended to a functor in [1].

An optional step consists in precomposing the obtained monad with a suitable functor from 'finite contexts' to 'general contexts', that is, with the inclusion from the comma category $J \downarrow \mathsf{sort}$ (where $J : \mathsf{Fin} \to \mathsf{Set}$ is the inclusion of finite sets into sets) to the slice category $\mathsf{Set}/\mathsf{sort}$. This would yield a monad *relative to* $\mathcal{J} : J \downarrow \mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$ [4].

The construction of 'term monads' from multi-sorted binding signatures presented here is one step in a larger project with Voevodsky, which aims to construct a C-system from the term monad of a $\{\mathsf{ty}, \mathsf{el}\}$-sorted binding signature [11].

References

[1] Benedikt Ahrens and Ralph Matthes. Heterogeneous substitution systems revisited. *To be published in the Postproceedings of TYPES'15, LIPIcs vol. 69.* http://arxiv.org/abs/1601.04299.

[2] Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. From signatures to monads in UniMath. 2016. https://arxiv.org/abs/1612.00693.

[3] Benedikt Ahrens and Julianna Zsidó. Initial Semantics for higher–order typed syntax in Coq. *Journal of Formalized Reasoning*, 4(1):25–69, September 2011.

[4] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1), 2015.

[5] Richard S. Bird and Ross Paterson. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.*, 9(1):77–91, 1999.

[6] Venanzio Capretta and Amy Felty. Higher-order abstract syntax in type theory. In *Logic Colloquium 2006*, volume 32 of *Lecture Notes in Logic*, pages 65–90. Cambridge U. Press, 2009.

[7] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract Syntax and Variable Binding. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 193–202, 1999.

[8] André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010.

[9] Alexander Kurz and Daniela Petrisan. On universal algebra over nominal sets. *Mathematical Structures in Computer Science*, 20(2):285–318, 2010.

[10] Ralph Matthes and Tarmo Uustalu. Substitution in non-wellfounded syntax with variable binding. *Theor. Comput. Sci.*, 327(1-2):155–174, 2004.

[11] Vladimir Voevodsky. C-system of a module over a Jf-relative monad. 2016. https://arxiv.org/abs/1602.00352.

Typing with Leftovers: a Mechanisation of ILL

Guillaume Allais[1]

Radboud University Nijmegen, The Netherlands gallais@cs.ru.nl

In a linear type system, all the resources available in the context have to be used exactly once by the term being checked. In traditional presentations of intuitionistic linear logic this is achieved by representing the context as a multiset which, in each rule, gets cut up and distributed among the premises. This is epitomised by the right rule for tensor (cf. Figure 1).

However, multisets are an intrinsically extensional notion and therefore quite arduous to work with in an intensional type theory. Various strategies can be applied to tackle this issue; most of them rely on using linked lists to represent contexts together with either extra inference rules to reorganise the context or a side condition to rules splitting the context so that it may be re-arranged on the fly. In the following example, $\approx$ stands for "bag-equivalence" of lists.

$$\frac{\Gamma \vdash \sigma \qquad \Delta \vdash \tau}{\Gamma, \Delta \vdash \sigma \otimes \tau}\otimes_i \qquad\qquad \frac{\Gamma \vdash \sigma \qquad \Delta \vdash \tau \qquad \Gamma, \Delta \approx \Theta}{\Theta \vdash \sigma \otimes \tau}\otimes_i$$

Figure 1: Introduction rules for $\otimes$. Left: usual presentation; Right: with reordering on the fly

All of these strategies are artefacts of the unfortunate mismatch between the ideal mathematical objects one wishes to model and their internal representation in the proof assistant. Short of having proper quotient types, this will continue to be an issue when dealing with multisets. The solution we offer tries not to replicate a set-theoretic approach in intuitionistic type theory but rather strives to find the type theoretical structures which can make the problem more tractable. Given the right abstractions, the proofs follow directly by structural induction.

McBride's recent work [2] on combining linear and dependent types highlights the distinction one can make between referring to a resource and actually consuming it. In the same spirit, rather than dispatching the available resources in the appropriate sub-derivations, we consider that a term is checked in a *given* context (typically named γ) on top of which usage annotations (typically named Γ, Δ, etc.) for each of its variables are super-imposed.

A derivation $\Gamma \vdash \sigma \boxtimes \Delta$ corresponds to a proof of σ in the underlying context γ with input (resp. output) usage annotations Γ (resp. Δ). Informally, the resources used to prove σ correspond to a subset of γ: they are precisely the ones which used to be marked *free* in the input usage annotation and come out marked *stale* in the *leftovers* Δ.

Wherever we used to split the context between sub-derivations, we now thread the leftovers from one to the next. Writing f_σ for a *fresh* resource of type σ and s_σ for a *stale* one, we can give new introduction rules for the variable with de Bruijn index zero, tensor and the linear implication, three examples of the treatment of context annotation, splitting and extension:

$$\frac{}{\Gamma \cdot f_\sigma \vdash \sigma \boxtimes \Gamma \cdot s_\sigma}var_0 \qquad \frac{\Gamma \vdash \sigma \boxtimes \Delta \qquad \Delta \vdash \tau \boxtimes \Theta}{\Gamma \vdash \sigma \otimes \tau \boxtimes \Theta}\otimes_i \qquad \frac{\Gamma \cdot f_\sigma \vdash \tau \boxtimes \Delta \cdot s_\sigma}{\Gamma \vdash \sigma \multimap \tau \boxtimes \Delta}\multimap_i$$

Figure 2: Introduction rules for var_0 $\otimes$ and $\multimap$ with leftovers

This approach is particularly well-suited to use intuitionistic linear logic as a type system for an untyped λ-calculus where well-scopedness is statically enforced: in the untyped calculus, it *is* the case that both branches of a pair live in the same scope. In our development, we use an inductive family in the style of Altenkirch and Reus [1] and opt for a bidirectional presentation [3] to minimise the amount of redundant information that needs to be stored.

Type Inference (resp. Type Checking) is then inferring (resp. checking) a term's type but *also* annotating the resources it consumed and returning the *leftovers*. These typing relations can be described by two mutual definitions; e.g. the definitions in Figure 2 would become:

$$\frac{}{\Gamma \cdot f_\sigma \vdash v_0 \in \sigma \boxtimes \Gamma \cdot s_\sigma} \qquad \frac{\Gamma \vdash \sigma \ni a \boxtimes \Delta \qquad \Delta \vdash \tau \ni b \boxtimes \Theta}{\Gamma \vdash \sigma \otimes \tau \ni (a, b) \boxtimes \Theta} \qquad \frac{\Gamma \cdot f_\sigma \vdash \tau \ni b \boxtimes \Delta \cdot s_\sigma}{\Gamma \vdash \sigma \multimap \tau \ni \lambda b \boxtimes \Delta}$$

Figure 3: Type *Inference* rule for var_0 and Type *Checking* rules for pairs and lambdas

For this mechanisation to be usable, it needs to be well-behaved with respect to the natural operations on the underlying calculus (renaming and substitution) but also encompass all of ILL. Our Agda formalisation (available at `https://github.com/gallais/typing-with-leftovers`) states and proves the following results for a system handling types of the shape:

$$\sigma, \tau, \ldots ::= \alpha \mid \mathbb{0} \mid \mathbb{1} \mid \sigma \multimap \tau \mid \sigma \oplus \tau \mid \sigma \otimes \tau \mid \sigma \,\&\, \tau$$

Lemma 1 (Framing)**.** *The usage annotation of resources left untouched by a typing derivation can be altered freely. The change is unnoticeable from the underlying λ-term's point of view.*

Lemma 2 (Weakening)**.** *The input and output contexts of a typing derivation can be expanded with arbitrarily many new resources. This corresponds to a weakening on the underlying λ-term.*

Lemma 3 (Parallel Substitution)**.** *Given a term and a typing derivation corresponding to each one of the fresh resources in its typing derivation's context, one can build a new typing derivation and a leftover environment. The corresponding action on the underlying λ-term is parallel substitution.*

Lemma 4 (Functional Relation)**.** *The typing relation is functional: for given "inputs", the outputs are uniquely determined. It is also the case that the input context is uniquely determined by the output one, the term and the type.*

Lemma 5 (Typechecking)**.** *Type checking (resp. Type inference) is decidable.*

Lemma 6 (Soundness)**.** *Typing derivations give rise to sequent proofs in ILL.*

Lemma 7 (Completeness)**.** *From a sequent proofs in ILL, one can build a pair of an untyped term together with the appropriate typing derivation.*

References

[1] Thorsten Altenkirch and Bernhard Reus. *Monadic Presentations of Lambda Terms Using Generalized Inductive Types*, pages 453–468. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[2] Conor McBride. *I Got Plenty o' Nuttin'*, pages 207–233. Springer International Publishing, 2016.

[3] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

From setoid hell to homotopy heaven?

Thorsten Altenkirch[1]*

School of Computer Science, University of Nottingham, UK

In my presentation I want to discuss the role of extensionality in Type Theory. Hence my contribution is not of a technical but of a partially historic and partially philosophical nature. It certainly expresses views which can and should be discussed.

To cope with the lack of extensional concepts in Intensional Type Theory, we used setoids: a type with an equivalence relation. This way we can supply the intended equality for function types and for quotient constructions like the reals. However, setoids have a number of disadvantages:

- Where exactly do we use setoids and where types?
- We have to introduce a lot of boilerplate, e.g. we define List as an operation on types but now we have to lift this also to setoids.
- This gets even worse when we consider families of setoids, as for example in categories where the objects have a non-trivial equality.
- We never actually hide the implementation, any user of a setoid may still depend on the implementation details.

To some degree this disadvantages can be overcome by discipline and by using additional tools but it seems to be more honest to built mechanisms into Type Theory itself.

While extensional principles are not provable in Intensional Type Theory on the other hand we are unable to observe intensional aspects of our definitions. This is different to set theory where we can ask intensional questions like $2 \in 3$? Or $\mathbb{N} \cap \mathrm{Bool} = \emptyset$? [1]

An alternative to Intensional Type Theory is Computational Type Theory [2] (Constable (2002)) which identifies judgemental equality and propositional equality. In Computational Type Theory functional extensionality is provable and quotient types are definable. Computational Type Theory is based on a set theoretic realizability semantics and it mimics set theoretic extensionality avoiding intensional questions.

Another option is to internalize the idea of setoids, if everything is a setoid what is the corresponding type theory? This was carried out in Altenkirch (1999); Altenkirch et al. (2007) leading to a system which we dubbed *Observational Type Theory.* This type theory is intensional in the sense that we don't have equality reflection but it supports extensional concepts like functional extensionality and quotients.

By propositional extensionality we mean that two propositions (that is types with at most one inhabitant) are equal if they are logically equivalent. This principle is true in set theory and can be added to Computational Type Theory and Observational Type Theory. However, why should we only stop at propositions? We can ask the question in general: when are two types equal.

*Supported by EPSRC grant EP/M016951/1 and USAF grant FA9550-16-1-0029.

[1]If we want to talk about intensional objects in Type Theory we can. E.g. we can represents sets as trees following Peterr Aczel and asks the above questions about those trees. However, in Type Theory we don't have to we can hide information which is not available in set theory.

[2]In the past this was called *Extensional Type Theory*

The answer is given by the univalence principle of Homotopy Type Theory which has propositional extensionality as a special case. In a nutshell univalence says that equality of types is equivalence which in the case of proposition boils down to logical equivalence and in the case of sets (types whose equality is propositional) to isomorphism.

An important consequence of univalence is that equality cannot be propositional. This can be easily seen because they are two isomporphisms of type $\mathrm{Bool} \simeq \mathrm{Bool}$ and identifying them is clearly inconsistent. Indeed we can construct arbitrarily complex types using only univalence and universes Kraus and Sattler (2015).

We cannot model this sort of equality in set theory and not in a system with equality reflection because here equality is proof-irrelevant by design. Our setoid approach is insufficient as well because we rely on equality being propositional to model Type Theory. However, it is clear how where to go, we need proof-relevant version of setoids where the equality has the same structure again, i.e. weak ω-groupoids. Recent work Cohen et al. (2016) shows that we can make this precise and not only model but also implement Type Theory using cubical sets.

Extensionality is essential if we want to construct towers of abstraction. We want to be able to replace any object by another one which behaves the same without the whole tower tumbling down. We need to reason upto extensional equality and not have to take into account the choice of encoding. Information hiding is essential once we move to large scale formalisation and for the time being Homotopy Type Theory is the only foundational theory which delivers extensionality for equality of types.

References

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

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-677-6. doi:http://doi.acm.org/10.1145/1292597.1292608.

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.

Robert L Constable. Naive computational type theory. In *Proof and System-Reliability*, pages 213–259. Springer, 2002.

Nicolai Kraus and Christian Sattler. Higher homotopies in a hierarchy of univalent universes. *ACM Trans. Comput. Logic*, 16(2):18:1–18:12, April 2015. ISSN 1529-3785. doi:10.1145/2729979. URL `http://doi.acm.org/10.1145/2729979`.

Type Theory with Weak J*

Thorsten Altenkirch[1], Paolo Capriotti[1], Thierry Coquand[2], Nils Anders Danielsson[2], Simon Huber[2], and Nicolai Kraus[1]

[1] University of Nottingham
[2] University of Gothenburg

Judgmental equality is central to intensional type theory, and closely related to computation. In typical formulations of intensional type theory two terms are judgmentally equal if they have identical normal forms. Computation tends to make it easier to write type-correct programs, because fewer explicit casts need to be inserted into the terms. Thus judgmental equalities can make type theories more convenient to use. Our central question is whether they also change the strength of the theories. What are the consequences of replacing some judgmental equalities by propositional (internal) equalities? Do we get a weaker theory? Is the most basic form of judgmental equality based on β-equality for functions more fundamental than stronger forms that also include computation for the J rule, or various forms of η-equality (for instance as presented by Allais, McBride, and Boutillier [1])? In this talk we do not provide any answers, but we record a few observations and a conjecture.

In extensional Martin-Löf type theory [6] any propositional equality can be turned into a judgmental equality. Hofmann [3] has shown that one variant of extensional type theory is a conservative extension of an intensional type theory, i.e. if a type in the intensional theory has an inhabitant in the extensional one, then a corresponding inhabitant exists in the intensional theory. A similar statement for the Calculus of (Inductive) Constructions is due to Oury [7]. In these settings one could thus say that the additional judgmental equalities do not add additional strength (except in so far as they allow the formation of new types).

We note that a very important assumption in both Hofmann's and Oury's setting is *uniqueness of identity proofs (UIP)*, a principle which is not always assumed, and which is even rejected in homotopy type theory. UIP can be derived in extensional type theory, but is independent of some forms of intensional type theory [4].

In the absence of UIP, an additional concept distinguishes judgmental and computational equality: *coherence*. Judgmental equality is some sort of *law*, while propositional equality is *data*, and data is not automatically well-behaved (for example, associativity in a bicategory is given by 2-morphisms, i.e. data, and coherence follows from the pentagon law).

To make one of the questions that we are interested in precise, consider a suitable version of Martin-Löf type theory with an equality type, written $x = y$, and without UIP. We assume function extensionality. For a type A, let us write I_A for the type $\Sigma_{x,y:A}\, x = y$. Disregarding universe levels, we can write down the type of the eliminator J as

$$\mathsf{J} : (A : \mathcal{U})\ (P : I_A \to \mathcal{U})\ (d : (x : A) \to P(x, x, \mathsf{refl}))\ (q : I_A) \to P(q). \tag{1}$$

Usually the β-rule for J is assumed to hold judgmentally: $\mathsf{J}^{A,P,d}(x, x, \mathsf{refl}) \equiv d(x)$. We ask ourselves what happens if we replace this rule by a postulated term J_β:

$$\mathsf{J}_\beta : (A : \mathcal{U})\ (P : I_A \to \mathcal{U})\ (d : (x : A) \to P(x, x, \mathsf{refl}))\ (x : A) \to \mathsf{J}^{A,P,d}(x, x, \mathsf{refl}) = d(x). \tag{2}$$

One reason why this might be interesting is that cubical type theory [2] has a type of paths that satisfies all of the identity type's axioms, except that the β-rule for J does not in general hold judgmentally.

*Altenkirch and Capriotti are supported by USAF grant FA9550-16-1-0029, Altenkirch and Kraus by EPSRC grant EP/M016951/1, and Danielsson by a grant from the Swedish Research Council (621-2013-4879).

Here is an example, first discussed in 2011 [5], that was originally intended to illustrate the lack of coherence that could arise. For simplicity we use subst (aka $\mathsf{transport}$), a non-dependent variant of J which is derivable from J. For a type A, a family $P : A \to \mathcal{U}$, and an equality $q : x = y$, we denote the term by $\mathsf{subst}^{A,P,q} : P(x) \to P(y)$. From J_β and function extensionality we can derive the equality $\mathsf{subst}_\beta^{A,P} : \mathsf{subst}^{A,P,\mathsf{refl}} = \mathsf{id}_{P(x)}$. Consider the two terms $\mathsf{subst}^{A,P,\mathsf{refl}}(\mathsf{subst}^{A,P,\mathsf{refl}}(p))$ and $\mathsf{subst}^{A,P,\mathsf{refl}}(p)$. There are two obvious ways to prove that the first term is equal to the second one: we can use $\mathsf{subst}_\beta^{A,P}$ to remove either the first or the second occurrence of $\mathsf{subst}^{A,P,\mathsf{refl}}$ in the first term. If J_β was a judgmental equality, then subst_β would just be (defined as) refl, and the two equalities between the terms would both be refl and thus be equal. In the version where J_β (and thus subst_β) is only given as a propositional equality, it is less clear whether the two equalities are equal. In fact, some of us believed for some time that they are not, and that the propositional J_β thus made the type theory weaker.

However, it turns out that the two equalities are equal. To see this, note that the pair $(\mathsf{subst}^{A,P,\mathsf{refl}}, \mathsf{subst}_\beta^{A,P})$ is an element of the *singleton type* $\Sigma_{f:P(x)\to P(x)} f = \mathsf{id}_{P(x)}$, and equal to the pair $(\mathsf{id}_{P(x)}, \mathsf{refl})$. With that replacement one can derive an equality between the two equalities.[1]

While our initial thought was that (2) is a postulated inhabitant of a not necessarily propositional type (i.e. a type which can have more than one element), and thus potentially problematic, we realised that we should consider (1) and (2) in combination—after all, J is a postulated constant as well. It turns out that the Σ-type of pairs $(\mathsf{J}, \mathsf{J}_\beta)$ is contractible, assuming that we already have instances of $(\mathsf{J}, \mathsf{J}_\beta)$ (for suitable universe levels).[1] Observe that, by exchange of Σ's and Π's, the Σ-type of pairs $(\mathsf{J}, \mathsf{J}_\beta)$ is equivalent to

$$\begin{aligned}&(A : \mathcal{U})\ (P : I_A \to \mathcal{U})\ (d : (x : A) \to P(x, x, \mathsf{refl})) \to \\ &\Sigma_{j:(q:I_A)\to P(q)}\ ((x : A) \to j(x, x, \mathsf{refl}) = d(x)).\end{aligned} \tag{3}$$

The function $\lambda x.(x, x, \mathsf{refl})$ is an equivalence between A and I_A, and using this equivalence in the second line of (3) one can see that this second line is equivalent to a singleton type. Thus the whole type (3) is contractible.

Hence, if we replace the usual judgmental computation rule for J by the constant J_β, then coherence issues do not seem to arise, and we conjecture that a conservativity result holds even in the absence of UIP.

References

[1] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: A sound and complete decision procedure, formalized. In *DTP'13*, 2013. doi:10.1145/2502409.2502411.

[2] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. Preprint arXiv:1611.02108v1 [cs.LO], 2016.

[3] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, U. Edinburgh, 1995.

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

[5] Nicolai Kraus. Homotopy type theory. Slides for a talk given at Nottingham's Functional Programming Lab Away Day, 2011. Available at `www.cs.nott.ac.uk/~psznk/docs/talk_away.pdf`.

[6] Per Martin-Löf. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics*, 104:153–175, 1982. doi:10.1016/S0049-237X(09)70189-2.

[7] Nicolas Oury. Extensionality in the calculus of constructions. In *TPHOLs 2005*, 2005. doi:10.1007/11541868_18.

[1]At the time of writing an Agda formalisation of a very similar statement is available at `http://www.cse.chalmers.se/~nad/listings/equality/README.Weak-J.html`.

Domain Theory in Type Theory via QIITs

Thorsten Altenkirch[1]*, Paolo Capriotti[1]†, and Bernhard Reus[2]

[1] University of Nottingham
[2] University of Sussex, Brighton

One way to express general recursive functions and non-terminating computations in type theory is to define a relevant part of domain theory with fixpoint theorems and reasoning principles. This has been attempted in various mechanised forms, e.g. Coq [4] or Lego [9]. While the latter works in the axiomatic setting of *Synthetic Domain theory* and implements lifting via an axiomatization of the Sierpinski space to define lifting as partial map classifier, the former follow Capretta's idea [5] of implementing lifting as an appropriate quotient of a coinductive definition of the delay monad. Benton et al. then define pointed ω-cpos as notion of domain and develop some domain theory and denotational semantics. Similarly, in [6] profinite domains are formalised in Coq together with the usual results regarding recursive domain equations. Dockins' approach, however, is not based on Capretta's delay monad but implements effective algebraic domains relying on the "innate 'effectiveness' of functions inside type theory" [6]. In all the above formalisations, fixpoints of mixed-variant domain equations $D \cong F(D, D)$, where F is a strict locally continuous bifunctor, are constructed as bilimits of $D_{n+1} = F(D_n, D_n)$ and the minimal invariance property [7, 8] is used to derive a reasoning principle (fixpoint induction) for the recursive domains obtained via those bilimits.

While the implementations of domain theory mentioned above demonstrate nicely how domains (and reasoning principles) can be "constructed" in type theory on top of the lifting monad, the authors of [4] argue that "some of the proofs and constructions are much more complex than they would be classically". Moreover, those formalisations of domain theory until now have been always carried out in *intensional type theory*. We argue that it is more convenient to work in *Homotopy Type Theory* and use the rather expressive concept of *higher inductive types* [10] that have been proposed based on homotopy-theoretic ideas. Even more conveniently, we can assume a trivial higher-dimensional structure and just work with the resulting types, called *quotient inductive-inductive types* (QIIT). Their semantics have been recently defined and investigated in [1] and used e.g. to give an elegant implementation of the syntax and semantics of basic dependent type theory [3]. QIITs allow us to introduce new constructors *and* equalities at the same time and therefore to implement the lift monad in a straightforward inductive fashion. In [2], this has been carried out for sets (flat cpos). The constructors for the cpo $(A_\perp, \sqsubseteq)$ are $\perp : A_\perp, \eta : A \to A_\perp, \mathrm{sup}$, and constructors stating that $\sqsubseteq$ is reflexive, transitive, antisymmetric, that $\perp \sqsubseteq x$, that sup is the least upper bound. In [2] it has already been shown that this QIIT is the usual lifting monad on sets and that it is equivalent to the coinductive construction from [4]. According to [2], this constitutes "a first step in the development of a form of constructive domain theory in type theory. It remains to be seen whether it is possible to, for instance, replicate the work of Benton et al. ...".

We prove this claim, i.e. that it is indeed possible (and appropriate) to define domain theory inside type theory by analyzing [2]'s lifting monad on pointed ω-cpos. The lifting monad can be factored into two adjunctions: one between cpos and pointed cpos (using QIITs) and one between cpos and sets (which just constructs the discrete cpo). The adjunction between pointed cpos and cpos also gives rise to the lifting comonad on pointed cpos which can be used in the construction of domain equations.

As example we implement streams over a (flat) set of data A, i.e. $\mathsf{S}(A) = (A_{\mathrm{flat}} \times \mathsf{S}(A))_\perp$ [1] by constructing $\mathsf{S}(A)$ as colimit of the diagram created by the functor $F(X) = (A_{\mathrm{flat}} \times X)_\perp$. The existence of positively defined recursive domains like this is demonstrated by constructing solutions as

*Supported by EPSRC grant EP/M016951/1 and USAF grant FA9550-16-1-0029.

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

[1]Here $(\_)_{\mathrm{flat}}$ denotes the embedding of sets into ω-cpos while the outermost $(\dots)_\perp$ is the lifting comonad.

ω-colimits in the usual way which are definable as a QIIT. On streams we can work out examples like the filter function filter: $\Pi_{A:\mathsf{Set}}(A \to 2) \to \mathsf{S}(A) \to \mathsf{S}(A)$ or the function that finds a specific element in a stream: $\mathsf{find} : \Pi_{A:\mathsf{Set}}(A \to 2) \to \mathsf{S}(A) \to A_\perp$. Those functions can be defined inductively on the argument stream. Other functions, like the sieve of Eratosthenes, can only be defined co-inductively. All necessary induction and coinduction principles can be derived as usual (e.g. structural induction or fixpoint induction).

In the formalisation, we use Martin-Löf (intensional) type theory with identity types, enriched by the extensionality principle for functions and the existence of quotient inductive-inductive types.

Our development validates the usefulness of quotient inductive-inductive types in extensional type theory. We summarise the similarities and differences with respect to "common" intensional constructive formulations of ω-cpos that use setoids as e.g. [4, 6]. One such aspect is, for instance, the smash product, $\otimes$, that caused problems in [4], requires an impredicative universe to be encoded in [9], but appears to be unproblematic in [6] which implements effective domains.

In the future, we would like to further extend the development to include mixed-variant functors, making use of the limit-colimit coincidence for ω-cpos, and to also investigate whether and how topological and synthetic domain theory, respectively, can be elegantly formalised in an extensional type theory with quotient inductive-inductive types. The fact that the Sierpinski space can be easily expressed as a higher inductive type [11] provides some hope in this direction.

The ultimate goal is to eventually produce a mechanisation of this constructive domain theory in Agda using postulates for the QIITs. Such a mechanisation will help us identify needs for appropriate tactics in proof construction. Our work thus generally acts as a sounding board for the development of quotient inductive-inductive types and corresponding reasoning principles.

References

[1] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. *CoRR*, abs/1612.02346, 2016.

[2] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited: The partiality monad as a quotient inductive-inductive type. In *FOSSACS*. Springer, 2017.

[3] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *ACM SIGPLAN Notices*, volume 51, pages 18–29. ACM, 2016.

[4] Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 115–130. Springer, 2009.

[5] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[6] Robert Dockins. Formalized, effective domain theory in Coq. In *International Conference on Interactive Theorem Proving*, pages 209–225. Springer, 2014.

[7] Peter J. Freyd. *Remarks on algebraically compact categories*, page 95106. London Mathematical Society Lecture Note Series. Cambridge University Press, 1992.

[8] Andrew M. Pitts. Relational properties of domains. *Inf. Comput.*, 127(2):66–90, 1996.

[9] Bernhard Reus. Formalizing synthetic domain theory. *Journal of Automated Reasoning*, 23(3):411–444, 1999.

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

[11] Niccoló Veltri. Partiality and non-termination in type theory. Slides for talk at EWSC'17, March 2017.

Normalisation by Evaluation for a Type Theory with Large Elimination

Thorsten Altenkirch[1] [*], Ambrus Kaposi[2], and András Kovács[2]

[1] University of Nottingham, Nottingham, United Kingdom
txa@cs.nott.ac.uk
[2] Eötvös Loránd University, Budapest, Hungary
{akaposi|andraskovacs}@caesar.elte.hu

We use normalisation by evaluation (NbE) to prove normalisation for a dependent type theory with a universe closed under dependent function space and booleans, where booleans are equipped with large elimination. This is the continuation of our previous work [2]. There, NbE is given for an intrinsically typed syntax with dependent function space and a base type. The main technical addition in the current work is the inclusion of quote and unquote in the model (similarly to the NbE proof for System F [1]). An Agda formalisation is work in progress.

The syntax

We define an intrinsically typed syntax for type theory, that is, it only contains well-typed terms. Conversion rules are added as equality constructors, yielding a higher inductive inductive type. Our metatheory has uniqueness of identity proofs, hence we call this definition quotient inductive inductive. Our sorts are contexts, types, substitutions and terms.

$$\mathsf{Con} : \mathsf{Set} \qquad \mathsf{Ty} : \mathsf{Con} \to \mathsf{Set} \qquad \mathsf{Tms} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set} \qquad \mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}$$

The core calculus is given by the following constructors, omitting the equalities expressing that Con and Tms form a category with terminal object $\cdot$. In the rule $,\circ$, $_*$ denotes transport.

$$\begin{array}{ll|ll}
\cdot & : \mathsf{Con} & -[-] & : \mathsf{Tm}\,\Delta\,A \to (\sigma : \mathsf{Tms}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma\,A[\sigma] \\
-,- & : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con} & \pi_2 & : (\sigma : \mathsf{Tms}\,\Gamma\,(\Delta, A)) \to \mathsf{Tm}\,\Gamma\,A[\pi_1\,\sigma] \\
-[-] & : \mathsf{Ty}\,\Delta \to \mathsf{Tms}\,\Gamma\,\Delta \to \mathsf{Ty}\,\Gamma & [\mathsf{id}] & : A[\mathsf{id}] \equiv A \\
\mathsf{id} & : \mathsf{Tms}\,\Gamma\,\Gamma & [][] & : A[\sigma][\nu] \equiv A[\sigma \circ \nu] \\
- \circ - & : \mathsf{Tms}\,\Theta\,\Delta \to \mathsf{Tms}\,\Gamma\,\Theta \to \mathsf{Tms}\,\Gamma\,\Delta & \pi_1\beta & : \pi_1\,(\sigma, t) \equiv \sigma \\
\epsilon & : \mathsf{Tms}\,\Gamma\,\cdot & \pi\eta & : (\pi_1\,\sigma, \pi_2\,\sigma) \equiv \sigma \\
-,- & : (\sigma : \mathsf{Tms}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma\,A[\sigma] \to \mathsf{Tms}\,\Gamma\,(\Delta, A) & ,\circ & : (\sigma, t) \circ \nu \equiv (\sigma \circ \nu), ({}_{[][]*}t[\nu]) \\
\pi_1 & : \mathsf{Tms}\,\Gamma\,(\Delta, A) \to \mathsf{Tms}\,\Gamma\,\Delta & \pi_2\beta & : \pi_2\,(\sigma, t) \equiv^{\pi_1\beta} t
\end{array}$$

We have a universe closed under Π and Bool, with small and large elimination for Bool. We omit listing the substitution laws and three out of four β laws for Bool.

$$\begin{array}{ll|l}
\mathsf{U} & : \mathsf{Ty}\,\Gamma & \mathsf{Bool} : \mathsf{Tm}\,\Gamma\,\mathsf{U} \\
\mathsf{El} & : \mathsf{Tm}\,\Gamma\,\mathsf{U} \to \mathsf{Ty}\,\Gamma & \mathsf{true}, \mathsf{false} : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\mathsf{Bool}) \\
\Pi & : (\hat{A} : \mathsf{Tm}\,\Gamma\,\mathsf{U}) \to \mathsf{Tm}\,(\Gamma, \mathsf{El}\,\hat{A})\,\mathsf{U} \to \mathsf{Tm}\,\Gamma\,\mathsf{U} & \mathsf{ifthenelse} : (\hat{C} : \mathsf{Tm}\,(\Gamma, \mathsf{El}\,\mathsf{Bool})\,U)(b : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\mathsf{Bool})) \\
\mathsf{lam} & : \mathsf{Tm}\,(\Gamma, \mathsf{El}\,\hat{A})\,(\mathsf{El}\,\hat{B}) \to \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,(\Pi\,\hat{A}\,\hat{B})) & \qquad \to \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\hat{C}[\mathsf{id}, \mathsf{true}]) \to \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\hat{C}[\mathsf{id}, \mathsf{false}]) \\
\mathsf{app} & : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,(\Pi\,\hat{A}\,\hat{B})) \to \mathsf{Tm}\,(\Gamma, \mathsf{El}\,\hat{A})\,(\mathsf{El}\,\hat{B}) & \qquad \to \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\hat{C}[\mathsf{id}, b]) \\
\Pi\beta & : \mathsf{app}\,(\mathsf{lam}\,t) \equiv t & \mathsf{true}\beta : \mathsf{ifthenelse}\,\hat{C}\,\mathsf{true}\,t\,f \equiv t \\
\Pi\eta & : \mathsf{lam}\,(\mathsf{app}\,t) \equiv t & \mathsf{Ifthenelse} : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\mathsf{Bool}) \to \mathsf{Tm}\,\Gamma\,\mathsf{U} \to \mathsf{Tm}\,\Gamma\,\mathsf{U} \to \mathsf{Tm}\,\Gamma\,\mathsf{U}
\end{array}$$

[*]Supported by EPSRC grant EP/M016951/1 and USAF grant FA9550-16-1-0029.

To define a function from the syntax, one needs to use the eliminator which requires a method for each constructor, including the equality constructors. This ensures that the function respects conversion (which is equality).

Normal forms

We define variables, neutral terms and normal forms as inductive predicates on terms.

$$\mathsf{isVar}, \mathsf{isNe}, \mathsf{isNf} : (\Gamma : \mathsf{Con})(A : \mathsf{Ty}\,\Gamma) \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Set}$$

Variables are de Bruijn indices. Neutral terms are either variables, applications or $\{\mathsf{i}|\mathsf{I}\}\mathsf{fthenelse}$ applied to a neutral boolean. Normal forms are either constructors (Π, lam, Bool, true, false) or neutral terms. We do not allow neutral terms of any type: the type has to be U, Bool or $\mathsf{El}\,A$ where A is neutral. This is to preserve uniqueness of normal forms for functions. Similarly we have $\mathsf{isNes} : (\Gamma, \Delta : \mathsf{Con}) \to \mathsf{Tms}\,\Gamma\,\Delta \to \mathsf{Set}$ for neutral substitutions.

Presheaf logical predicate

NbE works by evaluating the syntax in a model, then quoting semantic values back to normal forms. In our case, the model is a proof-relevant presheaf logical predicate. Contexts are interpreted as predicates on substitutions into that context, together with an unquote function. Types are mapped to predicates on their terms, bundled with quote and unquote functions. The interpretation of substitutions and terms provide the fundamental lemmas. All of this is stable under variable renamings (we call the category of renamings REN).

$$\begin{aligned}
\mathsf{P}_\Delta &: \forall\Psi.(\rho : \mathsf{Tms}\,\Psi\,\Delta) \to (\mathsf{r} : \mathsf{Set}) \times (\mathsf{u} : \mathsf{isNes}\,\Psi\,\Delta\,\rho \to \mathsf{r})\\
\mathsf{P}_A &: \forall\Psi.(\rho : \mathsf{Tms}\,\Psi\,\Delta) \to \mathsf{P}_\Gamma\,\Psi\,\rho.\mathsf{r} \to (t : \mathsf{Tm}\,\Psi\,A[\rho]) \to (\mathsf{r} : \mathsf{Set}) \times (\mathsf{q} : \mathsf{r} \to \mathsf{isNf}\,\Psi\,A[\rho]\,t)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \times (\mathsf{u} : \mathsf{isNe}\,\Psi\,A[\rho]\,t \to r)\\
\mathsf{P}_t &: \forall\Psi.(\rho : \mathsf{Tms}\,\Psi\,\Gamma)(q : \mathsf{P}_\Gamma\,\Psi\,\rho.\mathsf{r}) \to \mathsf{P}_A\,\Psi\,\rho\,q\,(t[\rho]).\mathsf{r}
\end{aligned}$$

The interpretation of the universe contains a predicate over codes. This predicate for a code A expresses that it is a normal form and that there is also a predicate on terms of type $\mathsf{El}\,A$ together with quote and unquote, again.

$$\begin{aligned}
&\mathsf{P}_\mathsf{U}\,\Psi\,(\rho : \mathsf{Tms}\,\Psi\,\Gamma)(p : \mathsf{P}_\Gamma\,\Psi\,\rho)(\hat{A} : \mathsf{Tm}\,\Psi\,\mathsf{U}).\mathsf{r}\\
&\qquad := \mathsf{isNf}\,\Psi\,\mathsf{U}\,\hat{A} \times \forall\Omega.(\beta : \mathsf{REN}(\Omega,\Psi))(t : \mathsf{Tm}\,\Omega\,(\mathsf{El}\,\hat{A}[\beta]))\\
&\qquad\qquad \to (\mathsf{r} : \mathsf{Set}) \times (\mathsf{q} : \mathsf{r} \to \mathsf{isNf}\,\Omega\,(\mathsf{El}\,\hat{A}[\beta])\,t) \times (\mathsf{u} : \mathsf{isNe}\,\Omega\,(\mathsf{El}\,\hat{A}[\beta])\,t \to \mathsf{r})
\end{aligned}$$

The predicate for Bool says for a term $b : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\mathsf{Bool})$ that $(b \equiv \mathsf{true}) + (b \equiv \mathsf{false}) + \mathsf{isNe}\,\Gamma\,(\mathsf{El}\,\mathsf{Bool})\,b$ (we don't have η for Bool).

Normalisation

Normalisation unquotes the interpretation of a term, also using the unquote of the identity substitution: $\mathsf{norm}\,(t : \mathsf{Tm}\,\Gamma\,A) := \mathsf{P}_A\,\mathsf{id}\,p\,t.\mathsf{q}\,(\mathsf{P}_t\,\Gamma\,\mathsf{id}_\Gamma\,p) : \mathsf{isNf}\,\Gamma\,A\,t$ where $p := \mathsf{P}_\Gamma\,\mathsf{id}_\Gamma.\mathsf{u}\,(\mathsf{idneu} : \mathsf{isNes}\,\Gamma\,\Gamma\,\mathsf{id}_\Gamma) : \mathsf{P}_\Gamma\,\Gamma\,\mathsf{id}_\Gamma.\mathsf{r}$.

References

[1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for system F. 1997.

[2] Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for dependent types. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, pages 6:1–6:16, 2016.

Monadic containers and Σ-universes

Thorsten Altenkirch[1*] and Gun Pinyo

School of Computer Science, University of Nottingham, UK

We present an observation which relates monadic containers and type theoretic universes closed under Σ-types and unit types (which we call Σ-universes).

A container Abbott et al. (2003) or polynomial functor Gambino and Hyland (2003) is given by a set of shapes $S : \mathbf{Set}$ and a family of positions $P : S \to \mathbf{Set}$ denoted as $S \lhd P$ which gives rise to an endofunctor on $\mathbf{Set}$ given by $[\![S \lhd P]\!]\, X = \Sigma s : S.P\, s \to X$ on objects. Morphisms between containers $S \lhd P$ and $T \lhd Q$ are given by a pair of a function on shapes $f : S \to T$ and a dependent function on positions $r : \Pi_{s:S} Q\,(f\, s) \to P\, s$ (note the change of direction) which gives rise to a natural transformation $[\![f \lhd r]\!]\,(s, p) = (f\, s, g \circ r_s)$, and $[\![-]\!]$ is a full and faithful functor from the category of containers to the category of endofunctors on $\mathbf{Set}$. A container is a monad if its associated functor is a monad.

A universe is given by a set of codes $\mathsf{U} : \mathbf{Set}$ and a family $\mathsf{El} : \mathsf{U} \to \mathbf{Set}$ which associates elements to names. We say that a universe has Σ-types if there are $\iota : \mathsf{U}$ and

$$\sigma : \Pi a : \mathsf{U}.(\mathsf{El}\, a \to \mathsf{U}) \to \mathsf{U}$$

such that

$$\begin{aligned} \mathsf{El}\, \iota &= 1 && (1)\\ \mathsf{El}\,(\sigma\, a\, b) &= \Sigma x : \mathsf{El}\, a.\mathsf{El}\,(b\, x) && (2) \end{aligned}$$

We define the non-dependent product as $a \otimes b = \sigma\, a\, (\lambda - .b)$. We also require:

$$\begin{aligned} a \otimes 1 &= a && (3)\\ 1 \otimes b &= b && (4)\\ \sigma\, a\, (\lambda x.\sigma\, (b\, x)\, (c x)) &= \sigma(\sigma\, a\, b)\, (\lambda x.c\, (\pi_0\, x)\, (\pi_1\, x)) && (5) \end{aligned}$$

The equations (3) - (5) are justified because the corresponding isomorphisms under El are valid. Borrowing categorical terminology we say that the universe is *lax*, if the equalities (1),(2) are replaced by functions:

$$\begin{aligned} &\mathsf{un} : \mathsf{El}\, \iota \to 1 && (6)\\ &\mathsf{pr}_{a\, b} : \mathsf{El}\, (\sigma\, a\, b) \to \Sigma x : \mathsf{El}\, a.\mathsf{El}\, (b\, x) && (7) \end{aligned}$$

Since the codomain of ι is 1, the 1st function exists anyway. We can restate equation (5) by composing the projections with pr.

Our main result is that lax Σ-universes are exactly monadic containers, choosing $\mathsf{U} \lhd \mathsf{El}$.

*Supported by EPSRC grant EP/M016951/1 and USAF grant FA9550-16-1-0029.

An example is List which is given by the Σ-universe with $\mathsf{U} = \mathbb{N}$ and $\mathsf{El} = \mathsf{Fin}$ (the family of finite sets where $0_n, 1_n, \ldots, (n-1)_n : \mathsf{Fin}\, n$). Now we show that it is closed under Σ-types and unit types; Here $\iota = 1$ and σ is defined as follows:

$$\begin{aligned} \sigma\, 0\, f &:\equiv 0 \\ \sigma (1+n)\, f &:\equiv f\, 0_n + \sigma\, n\, (f \circ (1+_n)) \end{aligned}$$

Intuitively, $\mathsf{El}\, \iota$ represents the unit type and $\mathsf{El}\, (\sigma\, n\, f)$ represents the sum type of $\mathsf{El}\, (f\, 0_n)$, $\mathsf{El}\, (f\, 1_n)$, ..., $\mathsf{El}\, (f\, (n-1)_n)$ respectively. The element of this sum type, in turn, can be sent to function pr in order to retrieve the original element of the corresponded type. For example, consider the case where $n = 3$ and $f = \lambda\{0_3 \mapsto 2, 1_3 \mapsto 4, 2_3 \mapsto 3\}$. Clearly $\sigma\, n\, f = 2+4+3 = 9$, so $5_9 : \mathsf{El}\, (\sigma\, n\, f)$, therefore, $\mathsf{pr}_{n\, f}\, 5_9 = (1, 2_4)$.

Proper Σ-universes correspond precisely to cartesian monadic containers. Another observation is that if we replace Σ by $\times$ we get exactly applicative functors. We hope that these observations shed a new light on monads and help deriving new reasoning principles for them. The free monad over a functor is given by the free Σ-universe over a given universe - this reproves a result in Gambino and Kock (2013).

Our work is clearly dual to the results in Ahman et al. (2012) but we need to understand the relationship better.

References

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 23–38, 2003. doi:10.1007/3-540-36576-1_2. URL `http://dx.doi.org/10.1007/3-540-36576-1_2`.

Danel Ahman, James Chapman, and Tarmo Uustalu. When is a container a comonad? In *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'12, pages 74–88, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28728-2. doi:10.1007/978-3-642-28729-9_5. URL `http://dx.doi.org/10.1007/978-3-642-28729-9_5`.

N. Gambino and J. Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154:153–192, January 2013. doi:10.1017/S0305004112000394.

Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In *International Workshop on Types for Proofs and Programs*, pages 210–225. Springer, 2003.

A Probabilistic Approach of Behavioural Types

Bogdan Aman and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science, Iaşi, România
bogdan.aman@iit.academiaromana-is.ro, gabriel@info.uaic.ro

In sequential systems, data types offer an effective basis of assuring a correct evolution. In distributed systems, the *behavioural types* were introduced to secure the compatibility of interaction patterns among processes [6]. The behavioural type of a process specifies its expected interactions by using a simple type language, and so determining a correct evolution.

Probabilities allows to describe the uncertainty in quantitative terms. Regarding the possible behaviours of a system, people working in artificial intelligence have used probability distributions over a restricted set of events, each of them corresponding to a certain type of situation. In such an approach, the probabilities assigned to behaviours are real numbers from $[0, 1]$ rather than the values 0 and 1. We adapt this idea to the framework of behavioural types.

An important feature of a probabilistic model is that it distinguishes between nondeterministic and probabilistic choices [5]. The nondeterministic choices refer to the choices made by an external process, while probabilistic choices are choices made internally by the process (not under control of an external process). Intuitively, a probabilistic choice is given by a set of alternative transitions, where each transition has a certain probability of being selected; moreover, for each choice the sum of all these probabilities is 1. Probabilistic extensions of various process calculi have been considered for distributed systems (e.g., see [5]). There are two possibilities of extending a model using probabilities: either to replace nondeterministic choices by probabilistic choices, or to allow both probabilistic choices and nondeterministic choices. Here we consider the second approach by allowing probabilistic choices made internally by the communicating processes (sending a value or a label), and also nondeterministic choices controlled by an external process (receiving a value or a label). It should be noticed that in our operational semantics we impose that for each received value/label, the continuation of a nondeterministic choice is unique; thus, the corresponding execution turns out to be completely deterministic.

We use a probabilistic extension of the process calculus presented in [7] for which we define and study a typing system which extends the behavioural multiparty session types by combining both nondeterministic and probabilistic behaviours. In defining a type system for such a calculus we get inspiration from the synchronous multiparty session types [1]. A natural way to define such an extension consists of adding probabilistic information to some of the actions, and adopting a mixture of the classical generative and reactive models [4].

This approach is novel among the existing models used to formalize communicating processes in the framework of multiparty session types, and is different from the one used when defining timed multiparty session types [2]. This is due to the fact that the time changes throughout the evolution of the systems, and thus the types should check if time is between certain given thresholds. Since in our approach the probabilities are static, the global types just should check if the probabilities to execute certain actions are exactly the desired ones.

We introduce a typing system with the purpose of typing efficiently the probabilistic behaviours. This typing system uses a map from shared names to either their sorts $(S, S', \ldots)$ or to a special sort $\langle G \rangle$ used to type the sessions. Since a type is inferred for each participating process in a session, we use the notation $T@q$ (called located type) to represent a local type T assigned to a participating process q. Using these, we define

$$\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, a : \langle G \rangle \mid \Gamma, X : \tilde{S}\tilde{T} \qquad \Delta ::= \emptyset \mid \Delta, \tilde{s} : \{T@q\}_{q \in I}.$$

A sorting $(\Gamma, \Gamma', \ldots)$ is a finite map from names to sorts, and from process variables to sequences of sorts and types. Typings $(\Delta, \Delta', \ldots)$ record linear usage of session channels by assigning a family of located types to a vector of session channels. We use the judgement $\Gamma \vdash P \triangleright \Delta$ saying that "under the environment Γ, process P has typing Δ".

We get some results dealing with the type preservation under equivalence and reduction. According to these results, if a well-typed process takes a reduction step of any kind, the resulting process will be also well-typed. This is due to the fact that in our setting, for any given well-formed process, at most one typing rule can be applied. This means that the shape of the typing derivation tree is unique and fully determined by the shape of the process.

Theorem 1. *$\Gamma \vdash P \triangleright \Delta$ and $P \to_{p_i} P'$ imply $\Gamma \vdash P' \triangleright \Delta'$, where $\Delta = \Delta'$ or $\Delta \Rightarrow_{p_i} \Delta'$.*

Theorem 2. *$\Gamma \vdash P \triangleright \Delta$ and $P \equiv P'$ imply $\Gamma \vdash P' \triangleright \Delta$.*

If $P \to_{p_1} P_1 \to_{p_2} P_2 \ldots \to_{p_k} Q$, then the probability to reach Q from P by using the path $p = p_1 * p_2 * \ldots * p_k$ is denoted by (P, p, Q). We define $prob(P, Q) = \sum_p (P, p, Q)$ as the probability to reach Q from P by considering all the possible paths between them. The set $FReach(P) = \{Q | P \to_r^* Q \text{ and } Q \not\to\}$ indicates the final reachable processes starting from P.

Theorem 3. *If P is a well-typed process s.t. $FReach(P) \neq \emptyset$, then* $\displaystyle\sum_{Q \in FReach(P)} prob(P, Q) = 1.$

As in [7], an *annotated* process P is the result of annotating the bound names of P. For instance, $(\nu a)P$ is annotated to become $(\nu a : \langle G \rangle)P$. These annotations are natural in our framework. Using them, we get an important result: given an annotated process P and a sorting Γ, it is decidable if there exists a typing Δ such that $\Gamma \vdash P \triangleright \Delta$. Moreover, if such a typing Δ exists, then there exists an algorithm to construct it.

Theorem 4. *Given an annotated process P and a sorting Γ, it is decidable if there exists a typing Δ s.t. $\Gamma \vdash P \triangleright \Delta$. If such a typing Δ exists, there is an algorithm to construct one.*

This approach preserves the classical type system, and additionally it satisfies the axioms of a probability theory for computing the probability of each behaviour. As far as we know, there is no other related work in behavioural types. Previously, a type system was added by one of the authors to a distributed π-calculus with timeouts to describe safe access permissions [3].

References

[1] A. Bejleri, N. Yoshida. Synchronous Multiparty Session Types. *Electronic Notes in Theoretical Computer Science* **241**, 3–33 (2009).

[2] L. Bocchi, W. Yang, N. Yoshida. Timed Multiparty Session Types. *Lecture Notes in Computer Science* **8704**, 419–434 (2014).

[3] G. Ciobanu, M. Koutny. Timed Migration and Interaction with Access Permissions. *Lecture Notes in Computer Science* **6664**, 293–307 (2011).

[4] R.J. van Glabbeek, S.A. Smolka, B. Steffen. Reactive, Generative and Stratified Models of Probabilistic Processes. *Information and Computation* **121**, 59–80 (1995).

[5] M. Herescu, C. Palamidessi. Probabilistic Asynchronous π-calculus. *Lecture Notes in Computer Science* **1784**, 146–160 (2000).

[6] K. Honda, V.T. Vasconcelos, M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. *Lecture Notes in Computer Science* **1381**, 22–138 (1998).

[7] K. Honda, N. Yoshida, M. Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM* **63**(1): article 9 (2016).

Bouncing threads for infinitary proof theory

David Baelde[1], Amina Doumane[2], Guilhem Jaber[3], and Alexis Saurin[2]

[1] LSV, ENS Cachan & Inria Paris, `dbaelde@ens-cachan.fr`,
[2] IRIF, CNRS & Univ. Paris Diderot, `{amina.doumane,alexis.saurin}@irif.fr`,
[3] Univ. Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP, `guilhem.jaber@ens-lyon.fr`

Fixed point logics are widely studied for their expressiveness, both from a model-theoretic and from a proof-theoretic viewpoint. Infinitary (and cyclic) proof systems have drawn increasing attention in recent years both from a proof-search and from a Curry-Howard perspective [7, 4, 6, 2]. Indeed, cyclic proof systems have a more natural proof search than finitary proof systems: one is free from the search of (co-)inductive invariants and cyclic proofs enjoy a true cut-admissibility result.

From a Curry-Howard point of view, circular proofs may be interesting since not only can they be more natural to write but they may also have richer computational behaviours. While cut-admissibility was known, procedural cut-elimination was only solved recently, first for the restricted additive fragment of LL only [6] and later for full $\mu\mathsf{MALL}^\infty$, an infinitary system based on multiplicative-additive linear logic with least and greatest fixed points [1] [3].

Most previous frameworks for circular and infinitary proofs (as well as tableaux methods) have very similar validity conditions. In the present abstract, we introduce a new validity condition for $\mu\mathsf{MALL}^\infty$, motivated by the dynamics of cut-elimination in circular proofs.

1 Motivation for bouncing threads

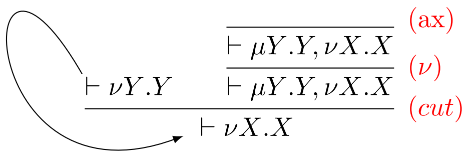


Figure 1: Example of a circular pre-proof.

In [3], a validity criterion for proofs of $\mu\mathsf{MALL}^\infty$ is introduced, extending that of [6]. We recall it briefly using the standard *immediate descendent* relation that can be found, for instance, in [5], denoted as $\sqsubseteq$ below. A $\mu\mathsf{MALL}^\infty$ proof is valid if any of its infinite branches is supported by a *valid thread* defined as follows. A thread t of an infinite branch $\gamma = (s_i)_{i\in\omega}$ formed by sequents s_i, is a sequence $(F_i)_{i\in\omega}$ of formula occurrences $F_i \in s_i$, s.t. $F_i \sqsubseteq F_{i+1}$. A *valid* thread t additionally satisfies that its minimal element (w.r.t the sub-formula ordering) among the set of formulas occurring infinitely often is a ν-formula, i.e. a greatest fixed-point.

It is then proved that a valid proof has a productive cut-elimination. However, there are proofs having a productive cut-elimination, whose cut-free normal forms are valid but which fall out of the scope of this validity criterion. This is particularly the case of proofs coming from the translation of a presentation of $\mu\mathsf{MALL}^\infty$ in natural deduction into the sequent calculus presentation. Indeed, when translating natural deductions, one typically gets the sequent calculus derivations of Figure 1.

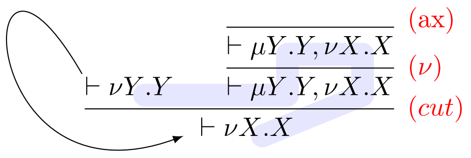


Figure 2: Example of a bouncing thread.

The previous pre-proof is not valid since it has no infinite threads; its infinite branch cannot be validated. This motivates the search for a validity condition with threads bouncing on axioms and cuts as in Figure 2.

2 The purely multiplicative case

First, we restrict our attention to MLL in which the cut-elimination and sequent proofs are well-known to enjoy better property (think, for instance, of the theory of proof nets).

Bouncing threads can be described as follows:

Definition 1. *A **pre-thread** is an infinite sequence of triples $(F_i, d_i, s_i)_{i\in\omega}$ (with $d_0 = \uparrow$) such that, for all i, we have $F_i \in s_i$, direction $d_i \in \{\uparrow, \downarrow\}$ and one of the following clauses holds:*

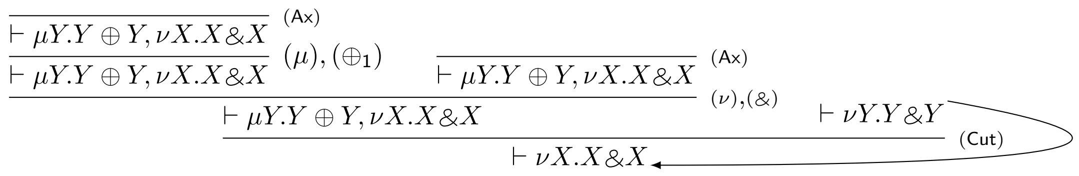
- *$d_i = d_{i+1} = \uparrow$, s_{i+1} is a premise of the rule of conclusion s_i, and $F_i \sqsubseteq F_{i+1}$;*
- *$d_i = d_{i+1} = \downarrow$, s_i is a premise of the rule of conclusion s_{i+1}, and $F_{i+1} \sqsubseteq F_i$;*
- *$d_i = \uparrow$, $d_{i+1} = \downarrow$ and $s_i = s_{i+1} = \{F_i, F_{i+1}\}$ is the conclusion of an axiom rule;*
- *$d_i = \downarrow$, $d_{i+1} = \uparrow$ and s_i and s_{i+1} are the two premises of the same cut rule, and $F_i = F_{i+1}^{\perp}$.*

*A thread is then a **persistent** pre-thread, in the sense of geometry of interaction/proof-nets, inducing a notion of visible/invisible formulas (or well-bracketed).*

We then say that a thread t is **valid** if the set of its visible formulas occurring infinitely often has a minimum (*wrt.* the subformula ordering) which is a ν-formula. A proof is said to be valid if all its infinite branches are supported by a valid thread. We can then prove that a valid proof has a productive cut-elimination procedure. To do so, we first eliminate the cuts forming the detours, so that we get a proof which is valid w.r.t. the criterion of [3].

3 Accommodating the additives

We shall not treat of the additive fragment here but simply point its complexity. With additive inferences, existence of a valid (bouncing) thread is not sufficient as shown below:



$\vdash \mu Y.Y \oplus Y, \nu X.X \& X$
(Ax)
$(\mu), (\oplus_1)$
$\vdash \mu Y.Y \oplus Y, \nu X.X \& X$
$\vdash \mu Y.Y \oplus Y, \nu X.X \& X$
(Ax)
$(\nu), (\&)$
$\vdash \mu Y.Y \oplus Y, \nu X.X \& X$
$\vdash \nu Y.Y \& Y$
(Cut)
$\vdash \nu X.X \& X$

This is due to additive contraction: intuitively, the only infinite branch of the above proof is duplicated infinitely many times and only some of these duplicates produce an infinite valid branch, some others are not productive (due to the μ inferences). To address this problem, we work slice by slice: each infinite branch of each slice shall contain a valid bouncing thread.

References

[1] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic (TOCL)*, 13(1):2, 2012.

[2] David Baelde, Amina Doumane, and Alexis Saurin. Least and greatest fixed points in ludics. In *24th EACSL Annual Conference on Computer Science Logic*, 2015.

[3] David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In *25th EACSL Annual Conference on Computer Science Logic*, 2016.

[4] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, December 2011.

[5] Samuel R Buss. An introduction to proof theory. *Handbook of proof theory*, 137:1–78, 1998.

[6] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: semantics and cut-elimination. In *Computer Science Logic 2013*, pages 248–262, 2013.

[7] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In *Foundations of Software Science and Computation Structures*, 2002.

The Clocks Are Ticking: No More Delays!
Reduction Semantics for Type Theory with Guarded Recursion

Patrick Bahr[1], Hans Bugge Grathwohl[2], and Rasmus Ejlers Møgelberg[1]

[1] IT University of Copenhagen
[2] Aarhus University

Abstract

Guarded recursion in the sense of Nakano allows general recursive types and terms to be added to type theory without breaking consistency. Recent work has demonstrated applications of guarded recursion such as programming with codata, reasoning about coinductive types, as well as constructing and reasoning about denotational models of general recursive types. As a step towards an implementation of a type theory with guarded recursion, we present Clocked Type Theory, a new type theory for guarded recursion that is more suitable for reduction semantics than the existing ones. We prove confluence, strong normalisation and canonicity for its reduction semantics, constructing the theoretical basis for a future implementation.

1 Introduction

Guarded recursion [4] allows recursion to be added to type theory without breaking consistency by introducing time steps in the form of a delay type modality $\triangleright$ (pronounced 'later'). Elements of type $\triangleright A$ are to be thought of as elements of type A only available one time step from now. Recursion arises from a fixed point operator mapping each productive endofunction (i.e. a function of type $\triangleright A \to A$) to its unique fixed point.

The most advanced type theory with guarded recursion to date is Guarded Dependent Type Theory (GDTT) [2], an extensional type theory with a notion of clocks each of which has a delay modality. Coinductive types can be encoded using guarded recursive types and universal quantification over clocks [1], which allows productivity to be expressed in types. In addition, GDTT has a notion of *delayed substitutions* allowing for coinductive, type theoretic reasoning about coinductive data and functions manipulating coinductive data. Delayed substitutions make it difficult to define a reduction semantics directly on GDTT. To solve this problem, we introduce a new type theory called Clocked Type Theory (CloTT), which can be seen as a refinement of GDTT.

2 Clocked Type Theory

Clocked Type Theory is an extension of dependent type theory with a special collection of sorts called *clocks*, which are inhabited by *ticks*. An assumption of the form $\alpha : \kappa$ states that α is assumed to be a tick on clock κ. In a context $\Gamma, \alpha : \kappa, \Gamma' \vdash$, tick variable α represents the assumption that a tick of time occurs on clock κ between the time when the values represented by the variables in Γ and those in Γ' are received. The delay modality $\triangleright$ is replaced by a form of dependent function type over clocks: The type $\triangleright(\alpha : \kappa).A$ is a type of suspended computations requiring a tick α on the clock κ to compute elements of type A. This can be understood as a dependent function type, with introduction and elimination rules given by abstraction over ticks, written $\lambda(\alpha : \kappa).t$, and application to ticks, written $t\,[\alpha]$. A term t can only be applied to a tick α' if all of the variables that t depend on are available before α', which corresponds to the intuition that t computes to a value of type $\triangleright(\alpha : \kappa).A$ before α'. We obtain the ordinary delay type modality by writing $\triangleright^\kappa A$ for $\triangleright(\alpha : \kappa).A$ if α is not free in A. The applicative functor laws for $\triangleright^\kappa$ follow from standard β and η rules for tick abstraction and application:

$$(\lambda(\alpha' : \kappa).t)\,[\alpha] \to t\,[\alpha/\alpha'] \qquad\qquad \lambda(\alpha : \kappa).(t\,[\alpha]) \to t \quad \text{if } \alpha \notin \mathsf{fv}(t)$$

A suspended computation represented by a closed term of type $\triangleright(\alpha : \kappa).A$ can be forced by applying it to the tick constant $\diamond$. In general, this is unsafe for open terms, since it breaks the productivity guarantees that the typing system should provide. For example, the term $\lambda(x : \triangleright^\kappa A).x\,[\diamond]$ should not be typeable, because it is not productive. It is safe, however, to force an open term if the clock κ does not appear free in the context Γ.

A term $f : \triangleright^\kappa A \to A$ is a productive function taking suspended computations of type A and returning values of type A. The *delayed fixed point* $\mathsf{dfix}^\kappa f$ of f is an element of type $\triangleright^\kappa A$ which, when given a tick, applies f to itself. Crucially, $\mathsf{dfix}^\kappa f$ only unfolds in the reduction semantics if applied to the tick constant, i.e. we have the reduction rule $(\mathsf{dfix}^\kappa\, t)\,[\diamond] \to t\,(\mathsf{dfix}^\kappa\, t)$. In particular, any term $(\mathsf{dfix}^\kappa\, t)\,[\alpha]$ remains stuck if α is a tick variable.

3 Results

We argue that CloTT is at least as expressive as the fragment of GDTT without identity types, by giving a translation of the latter into the former. The translation maps most of the equational rules of GDTT to equalities that follow from the reduction semantics of CloTT. In particular, most of the rules for delayed substitutions follow in fact from the β and η rules for tick abstraction and standard rules for substitutions. Some of the equational rules of GDTT do not follow from the reduction semantics, but we argue that these are most naturally expressed in CloTT as propositional equalities stating that ‘all ticks are equal’ and ‘all clocks are equal’. We argue informally why these can be added to a future extension of CloTT with path types (in the sense of Cubical Type Theory [3]) without breaking canonicity.

Our main results concern the reduction semantics of CloTT, which we show is confluent and strongly normalising. As a consequence of this, equality of terms and types can be decided by reducing these to their unique normal forms. This decision procedure is a major step towards a type checking algorithm for CloTT. We also prove a canonicity theorem stating that every closed term of type Nat reduces to a natural number. As a consequence of this we derive the statement of productivity: Given a well typed closed term of stream type, its n’th element can be computed in finite time. This is a formal statement of the fact that guarded recursion captures productivity of coinductive definitions in types.

References

[1] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 197–208. ACM, 2013.

[2] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In *FOSSACS*, 2016.

[3] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[4] Hiroshi Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

Regularity for Free

Thibaut Balabonski[1], Marco Gaboardi[2], Chantal Keller[1], and Benoît Valiron[1]

[1] LRI, CNRS, Univ. Paris-Sud, CentraleSupélec, Université Paris-Saclay, France
[2] University at Buffalo, US

The goal of software testing is to guarantee that a program will behave according to a specification by confronting on a well-chosen set of test data its behavior against the specified one. In order to maximize the effectiveness of a given test data, special care must be taken to ensure that it covers all, or most, of the possible behaviors of the program: this is known as *code coverage* [5].

A recent line of work considers the question from a formal standpoint, in the context of functional programming and inductive datatypes. In this work, one often assumes the *regularity hypothesis*, that informally amounts to testing programs on inputs up to a certain size [1, 3].

As an example, to test that a program P taking a list as an input is equivalent to some reference program Q, one often limits to bounded lists up to a certain length k, and assumes the following regularity hypothesis:

$$(\forall l : {'a}\,\mathrm{list}_k \cdot P(l) = Q(l)) \Rightarrow (\forall l : {'a}\,\mathrm{list} \cdot P(l) = Q(l))$$

where ${'a}\,\mathrm{list}_k$ is the type of polymorphic lists of length up to k. This means: if P and Q behave similarly for all lists of size less or equal to k, then P and Q behave the same on all lists.

This approach allows one to bridge a link between test and verification [1]. Indeed, a (complete) proof that a program meets its specification is (1) a proof of the regularity hypothesis, and (2) the fact that the program successfully passes the tests. Step (2) is only a matter of computation: running the tests. Step (1) is more involved: it requires (a) to exhibit a bound k rendering the regularity hypothesis valid, which is not decidable in general, and (b) to prove the regularity hypothesis given the candidate bound k.

In this short note, we hint at one possible way to solve steps (a) and (b) in a unified manner: the use of a dedicated type system. The technique we use is based on a line of work originated from bounded linear logic [4].

A small language of natural numbers. As a first step, we consider a simply-typed lambda-calculus with a base type $\mathbb{N}$ of natural numbers, constructors $\bar{n}$ for each natural numbers and infix binary operators "+" and "$*$". The language is formally defined as

$$\begin{array}{lcl} M, N & ::= & x \mid \lambda x.M \mid MN \mid M + N \mid M * N \mid \bar{n} \\ A, B & ::= & \mathbb{N} \mid A \to B \end{array}$$

with the usual notion of typing context, judgement and the typing rules one would expect. The operational semantics of the language is given through a straightforward beta-reduction; to each closed term P of type $\mathbb{N}$ one can associate a single $\bar{n}_P$ to which it eventually reduces to. Under this equivalence, a typing judgement $x : \mathbb{N} \vdash P : \mathbb{N}$ corresponds to a function $f_P : \mathbb{N} \to \mathbb{N}$.

We now turn to the question of the kind of properties we want to test. In this simple framework we consider two typing judgements $x : \mathbb{N} \vdash P : \mathbb{N}$ and $x : \mathbb{N} \vdash Q : \mathbb{N}$, and we ask whether f_P and f_Q are pointwise the same function. We want a number k of tests guaranteeing that whenever the two functions are equal on k distinct inputs, then they are equal everywhere.

We claim that in this case the question can be solved using a slightly more evolved type system.

$$\frac{}{x : !_1 A \vdash x : A} \quad \frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \multimap B} \quad \frac{\Delta \vdash M : A \multimap B \quad \Psi \vdash N : A}{\Delta + \Psi \vdash MN : B} \quad \frac{\Delta \vdash M : B \quad x \notin |\Delta|}{\Delta, x : !_0 A \vdash M : B}$$

$$\frac{}{\vdash \bar{n} : \mathbb{N}} \quad \frac{\Delta \vdash M : \mathbb{N} \quad \Psi \vdash N : \mathbb{N}}{\max(\Delta, \Psi) \vdash M + N : \mathbb{N}} \quad \frac{\Delta \vdash M : \mathbb{N} \quad \Psi \vdash N : \mathbb{N}}{\Delta + \Psi \vdash M * N : \mathbb{N}}$$

Table 1: Typing derivations

An indexed type system. Following the spirit of bounded linear logic, we define a new type system with annotations of the form

$$A \quad ::= \quad \mathbb{N} \,|\, A \multimap B \,|\, !_n A$$

where n ranges over natural numbers. The typing derivations are described in Table 1. In the rules for application, sum and product, we assume that Δ and Ψ are respectively of the form $x_1 : !_{r_1} A_1, \dots, x_n : !_{r_n} A_n$ and $x_1 : !_{s_1} A_1, \dots, x_n : !_{s_n} A_n$ (if some variable y is missing on one side, for the purpose of the computation we silently add it with a type $y : !_0 B$). Then $\Delta + \Psi$ is $x_1 : !_{r_1+s_1} A_1, \dots, x_n : !_{r_n+s_n} A_n$ and $\max(\Delta, \Psi)$ is $x_1 : !_{\max(r_1,s_1)} A_1, \dots, x_n : !_{\max(r_n,s_n)} A_n$. Note that any valid derivation in this indexed type system yields a valid derivation in the simply-typed version. In particular, if $\Delta \vdash M : \mathbb{N}$ is valid, the function f_M can be defined through this correspondance.

Theorem. If $x_1 : !_{r_1}\mathbb{N} \dots x_n : !_{r_n}\mathbb{N} \vdash M : \mathbb{N}$ then f_M is a polynomial on the variables $x_1, \dots,$ x_n. The degree of the polynomial in each variable x_i is r_i. □

Now, if $x : \mathbb{N} \vdash P : \mathbb{N}$ and $x : \mathbb{N} \vdash Q : \mathbb{N}$, to decide if $f_P = f_Q$ it is enough to

1. Index both P and Q. From the theorem f_P and f_Q are polynomials in x and the indexation gives us bounds d_P and d_Q for the degree of each polynomial.

2. Test equality of $P[x := v_i]$ and $Q[x := v_i]$ for $\max(d_P, d_Q)$ distinct values v_i's, using the principal theorem of polynomial arithmetics.

For this toy language we are therefore able to reconciliate tests and proof: an algorithm gives us a number of tests to perform, equivalent to the decidability of polynomial equality. This approach extends straightforwardly to multivariate polynomials.

Perspectives. Of course the proposed language is very constrained, and we want to extend it to more language constructs: tests, iterations… Unfortunately, most interesting extensions breaks the main reason for which we are able to link test and proof: the principal theorem of polynomial arithmetics. Extension of the language usually changes the semantics of programs to more exotic form of functions, and the theorem is no longer valid, cutting the link to finish the proof.

It is as of yet an active area of research to figure out whether we can backup this seemingly simple procedure using an argument independant from polynomial arithmetic.

This project is the first step towards a deeper and more formal understanding of regularity and code coverage. Code coverage is nowadays the standard used by industrials to decide on a testing technique [7, 8]. Having a better, formal comprehension of design methods of tests with good properties is therefore interesting not only from an academic perspective, but also towards a more concrete usage.

Bibliography.

[1] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Asp. Comp.* 25(5): 683-721 (2013)

[2] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for lambda-terms. *Archiv für Mathematische Logik und Grundlagenforschung.* 19:139156 (1978).

[3] Marie-Claude Gaudel. Testing can be formal, too. *TAPSOFT'95*, LNCS vol 915, 82-96 (1995).

[4] Jean-Yves Girard, Andre Scedrov and Philip J. Scott. Bounded linear logic. *Th. Comp. S.* 97(1):1-66 (1992)

[5] Joan C. Miller and Clifford J. Maloney. 1963. Systematic mistake analysis of digital computer programs. *Commun. ACM* 6(2): 58-63 (1963).

[6] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. *SIGPLAN Not.* 45(9):157-168 (2010)

[7] DO-178B, Software Considerations in Airborne Systems and Equipment Certification.

[8] ISO 26262-6:2011 *Road vehicles – Functional safety – Part 6: Product development at the software level.*

A modular formalization of type theory in Coq

Andrej Bauer[1][*], Philipp G. Haselwarter[2][†], and Théo Winterhalter[3]

[1] University of Ljubljana, Slovenia
Andrej.Bauer@andrej.com
[2] University of Ljubljana, Slovenia
philipp@haselwarter.org
[3] École Normale Supérieure Paris-Saclay, France
theo.winterhalter@ens-cachan.fr

Abstract

We present a complete formalization of type theory in the Coq proof assistant. We use the type class mechanism to break down the formalization into fragments that can then be combined to several variants of type theory. We prove basic meta-theorems guaranteeing that our formulation is complete and sensible, as well as theorems that facilitate applications. To test the library, we formalize a translation of type theory into type theory by Boulier et al. which shows that function extensionality is not derivable.

Introduction. As anyone who has ever proved meta-theorems about dependent type theory will attest, the excitement of discovering a new fact is dulled by the drudgery of checking large boring proofs. In fact, just *stating* precisely and completely all the rules of type theory is considered inhumane, and therefore omitted in practically all presentations of type theory. The received wisdom says that such informal rigor causes no harm and that everything is known to work out. This may well be the case—although how can we really know?—but we have learned the hard way that incomplete formulations of meta-theorems with sloppy proofs lure us into a morass of confusion and half-truths that hinders progress. In other words, we shall not trust ourselves again to have stated a meta-theorem about dependent type theory, and much less proved it, unless it is fully formalized. Our formalization of type theory is implemented in the Coq proof assistant [3], and is freely available at [1].

The basic setup. For our purposes (which have not come to fruition yet), we needed a formalization of several variants of type theory, which all shared common basic structure but differed in the inclusion or exclusion of certain principles and constructors. We intended to manipulate all parts of the syntax and the derivations, including substitutions and judgmental equalities. We thus opted for a "deep" formalization of the syntax and the rules as straightforward inductive types, thereby having complete control over the object-level theory, albeit at the price of having to perform manually various syntactic manipulations that a "shallower" embedding would simply pass on to the ambient type theory of the proof assistant. After several trials and make-overs we opted for a nameless representation of variables, explicit substitutions, and terms fully annotated with types.

At present the formalization includes product and identity types, base types (the empty type, the unit type, and booleans), simple products, a hierarchy of Tarski universes, and an impredicative universe of propositions. The modular design ought to make it easy to add other constructions.

[*]The author acknowledges the financial support from the Slovenian Research Agency (research core funding No. P1-0294) at Institute of Mathematics, Physics and Mechanics, Ljubljana, Slovenia.

[†]This material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.

Paranoid and economic inference rules. There is an amount of freedom in the formulation of inference rules. For example, the introduction rule for the identity type can be stated

$$\text{as} \quad \frac{\Gamma\ \mathsf{context} \qquad \Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{refl}_A\, u : \mathsf{Id}_A(u,u)} \quad \text{or as} \quad \frac{\Gamma \vdash u : A}{\Gamma \vdash \mathsf{refl}_A\, u : \mathsf{Id}_A(u,u)}.$$

We call the left one *paranoid* and the right one *economic*. When proving facts about type theory, it is generally preferable to have the paranoid version as an assumption and the economic one as the goal. We formulated type theory so that the paranoid or the economic variant of the rules can be chosen easily through a type class instance, and we proved meta-theorems showing that they derive the same judgments. Thus, we can switch between them at ease and use whichever is more convenient in a given situation.

Configurations and variants of type theory. The inductive types for the judgment forms are parameterized by type classes that either enable or disable various features of the formalization. By providing instances of the type classes, the user may combine the features to their liking. It is also possible to leave a feature unconfigured, and thus have a development which is agnostic with respect to it. At present we support configurability of the following features: paranoid vs. economic rules, the η-rule for functions, equality reflection, identity types, simple products, base types, a hierarchy of Tarski universes, and an impredicative universe of propositions.

Sanity theorems and other meta-theorems. In our experience it is easy to forget an inference rule, or just formulate it badly. We proved several meta-theorems which instill some trust in our formulation, and are generally useful to have:

- *sanity* theorems stating, e.g., that if $\Gamma \vdash u : A$ then $\Gamma \vdash A\ \mathsf{type}$ and $\Gamma\ \mathsf{context}$,
- the paranoid and the economic variants derive the same judgments,
- uniqueness of typing: if $\Gamma \vdash u : A$ and $\Gamma \vdash u : B$ then A and B are judgmentally equal.

Another desirable theorem (which we intend to prove) is elimination of explicit substitutions: every term is judgmentally equal to one without substitutions. A proof of this fact amounts to computation of substitutions at the meta-level.

Conclusion. To test the viability of our development we used it to state a translation of type theory into type theory which invalidates function extensionality, following the work of S. Boulier, P.-M. Pédrot, and N. Tabareau [2]. Our original motivation was formalization of elimination of equality reflection, but we leave that for future work.

There are several ways in which the library can be improved: we can provide better tactics for constructing terms and derivations, improve the syntax and notations, and include more features, such as inductive types and other type formers.

References

[1] Andrej Bauer, Philipp G. Haselwarter, and Théo Winterhalter. The 'formal-type-theory' repository. Available at `https://github.com/TheoWinterhalter/formal-type-theory/tree/types-2017`.

[2] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Certified Programs and Proofs – CPP 2017*, pages 182–194, January 2017.

[3] The Coq development team. *The Coq proof assistant reference manual*, 2017. Version 8.6, available at `http://coq.inria.fr`.

Presheaf semantics for guarded dependent type theory

Aleš Bizjak[1] and Rasmus Ejlers Møgelberg[2]

[1] Aarhus University (abizjak@cs.au.dk)
[2] IT University of Copenhagen (mogel@itu.dk)

Guarded recursion [10] is a form of recursion in which the circularities of recursive definitions are broken using time-steps encoded by a type modality. Guarded Dependent Type Theory (GDTT) [4] is a type theory with guarded recursive types and terms. This has e.g., been used [9] to model the programming language FPC by modelling recursive types as guarded recursive types, and to prove computational adequacy of this interpretation inside GDTT also using guarded recursion. It is expected that also the many recent applications of guarded recursion to logics for programming languages with advanced features such as general references, recursive types, countable non-determinism, and concurrency [2, 3, 11] can be formalised in GDTT.

Following an idea of Atkey and McBride [1], guarded recursion can also be used for programming with coinductive data using a notion of clocks and universal quantification over these. For example, if A is a type, and κ is a clock variable fresh for A, the recursive type $\mathsf{Str}^{\kappa}_{g}(A) \cong A \times \blacktriangleright^{\kappa} \mathsf{Str}^{\kappa}_{g}(A)$ exists in GDTT and can be thought of as a type of streams of A's whose head is immediately available, but whose tail takes a κ-time step to compute. Using a fixed point operator $\mathsf{fix}^{\kappa}_{X} : (\blacktriangleright^{\kappa} X \to X) \to X$ one can e.g. define a constant stream of $a : A$ as $\mathsf{fix}^{\kappa}_{\mathsf{Str}^{\kappa}_{g}(A)}(\lambda xs.\, \langle a, xs\rangle)$. The type $\forall\kappa.\mathsf{Str}^{\kappa}_{g}(A)$ is then a coinductive type of streams [1, 8], and it is known [1] that a wide range of operations on coinductive types can be expressed in this framework, encoding productivity in types using guarded recursion. In GDTT one can also do coinductive reasoning using guarded recursion and prove e.g. bisimilarity of streams [4].

Denotational semantics

The single clock case of guarded recursion can be modelled in the topos of trees [2], i.e., by modelling a type as a family of sets $(X_n)_{n\in\mathbb{N}}$ indexed by natural numbers together with restriction maps of type $X_{n+1} \to X_n$ for all n. In previous work [5] we have shown how to extend this to define a family of presheaf categories $\mathsf{GR}\,(\Delta)$ indexed over clock contexts (finite sets of clock variables) Δ. This family essentially carries enough structure to model GDTT, but constructing the model in practice is complicated by the fact that the morphisms $\mathsf{GR}\,(\Delta) \to \mathsf{GR}\,(\Delta')$ corresponding to substitutions in the clock context, only preserve structure up to isomorphism.

In this talk we present a different solution. We give a single presheaf category $\mathsf{Psh}(\mathbb{T}^{\mathsf{op}})$ in which one can construct a model of GDTT by extending existing techniques for modelling type theory in presheaf categories. The underlying category $\mathbb{T}$ has as objects pairs (Δ, δ), where Δ is a clock context and $\delta : \Delta \to \mathbb{N}$ is a map. A morphism from (Δ, δ) to (Δ', δ') is a map $\sigma : \Delta \to \Delta'$ such that $\delta'\sigma \leq \delta$ in the pointwise order. There is a presheaf of clocks $\mathcal{C}$ defined as $\mathcal{C}(\Delta, \delta) = \Delta$ (which extends to morphisms because we consider covariant presheaves on $\mathbb{T}$), which can be used to interpret clock variable assumptions of the form $\kappa : \mathsf{clock}$ in a context. Universal quantification over clocks $\forall\kappa.A$ is interpreted as if it were an ordinary dependent product, i.e., as $\Pi\kappa : \mathsf{clock}.A$.

Orthogonality

The GDTT axiom of *clock irrelevance* states essentially that if κ is fresh for A then the map $A \to \forall\kappa.A$ mapping x to $\Lambda\kappa.x$ (where Λ is abstraction over clock variables) is an isomorphism.

This axiom is crucial for the encoding of coinductive types: For $\forall\kappa.\mathsf{Str}^{\kappa}_{g}(A)$ to be a coinductive type of streams of A's in the model, we must at least have $\forall\kappa.\mathsf{Str}^{\kappa}_{g}(A) \cong A \times \forall\kappa.\mathsf{Str}^{\kappa}_{g}(A)$. To construct this isomorphism we use the fact that $\forall\kappa.\blacktriangleright^{\kappa} X \cong \forall\kappa.X$ in GDTT and clock irrelevance to show

$$\begin{aligned}\forall\kappa.\mathsf{Str}^{\kappa}_{g}(A) &\cong \forall\kappa.A \times \blacktriangleright^{\kappa} \mathsf{Str}^{\kappa}_{g}(A)\\ &\cong (\forall\kappa.A) \times (\forall\kappa.\blacktriangleright^{\kappa} \mathsf{Str}^{\kappa}_{g}(A))\\ &\cong A \times \forall\kappa.\mathsf{Str}^{\kappa}_{g}(A)\end{aligned}$$

Since quantification over clocks is a dependent function type over $\mathcal{C}$ in our model, the axiom of clock invariance simply states that the constant function map $A \to A^{\mathcal{C}}$ is an isomorphism. We say that a presheaf A is *orthogonal* to $\mathcal{C}$ if this holds, and it is an invariance of the model construction that the interpretation of a type is orthogonal to $\mathcal{C}$. For dependent types modelled as maps $A \to B$, the requirement becomes a unique lifting property wrt all projections of the form $\Gamma \times \mathcal{C} \to \Gamma$ as indicated in the diagram below

$$\begin{tikzcd}\Gamma \times \mathcal{C} \arrow[r] \arrow[d] & A \arrow[d] \\ \Gamma \arrow[r] \arrow[ur, dotted, "\exists!"] & B\end{tikzcd}$$

Since $\mathcal{C}$ is not orthogonal to itself, clock cannot be a type. This is similar to the situation in cubical type theory [6], where the interval is not fibred, and so not a type.

If there is time, the talk will also cover universes. The standard Hofmann-Streicher universe [7] in $\mathsf{Psh}(\mathbb{T}^{\mathsf{op}})$ is not orthogonal to $\mathcal{C}$, and so not a type. Instead, GDTT has a family of universes U_{Δ} indexed by clock contexts Δ.

References

[1] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208. ACM, 2013.

[2] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012.

[3] A. Bizjak, L. Birkedal, and M. Miculan. A model of countable nondeterminism in guarded type theory. In *RTA-TLCA*, pages 108–123, 2014.

[4] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In *FOSSACS*, pages 20–35, 2016.

[5] A. Bizjak and R. E. Møgelberg. A model of guarded recursion with clock synchronisation. *Electr. Notes Theor. Comput. Sci.*, 319:83–101, 2015.

[6] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[7] M. Hofmann and T. Streicher. Lifting Grothendieck universes. Unpublished, 1999.

[8] R. E. Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Proceedings of CSL-LICS 2014*, pages 71:1–71:10. ACM, 2014.

[9] R. E. Møgelberg and M. Paviotti. Denotational semantics of recursive types in synthetic guarded domain theory. In *LICS*, pages 317–326, 2016.

[10] H. Nakano. A modality for recursion. In *LICS*, pages 255–266. IEEE, 2000.

[11] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.

An interpretation of system F through bar recursion*

Valentin Blot

Queen Mary University of London

Abstract

There are two possible computational interpretations of second-order arithmetic: Girard's system F and Spector's bar recursion. While the logic is the same, the programs obtained from these two interpretations have a fundamentally different computational behavior and their relationship is not well understood. We make a step towards a comparison by defining the first translation of system F into a simply-typed total language with bar recursion. This translation relies on a realizability interpretation of second-order arithmetic. Due to Gödel's incompleteness theorem there is no proof of termination of system F within second-order arithmetic. However, for each individual term of system F there is a proof in second-order arithmetic that it terminates, with its realizability interpretation providing a bound on the number of reduction steps to reach a normal form. Using this bound, we compute the normal form through primitive recursion. Moreover, since the normalization proof of system F proceeds by induction on typing derivations, the translation is compositional. The flexibility of our method opens the possibility of getting a more direct translation that will provide an alternative approach to the study of polymorphism, namely through bar recursion.

Second-order λ-calculus [Gir71, Rey74] is a poweful type system allowing us to type terms such as $\lambda x.x\,x$. The language obtained is still strongly normalizing, but all the proofs of this fact so far have been relying on the notion of reducibility candidates (RCs): sets of λ-terms satisfying some axioms. In these proofs, every type has an associated RC such that every term belongs to the RC associated to its type. Then the normalization property follows as a consequence of the axioms of RCs. An important aspect of these proofs is their impredicativity: the RC associated to a universally quantified type is obtained as an intersection over all RCs. The translation presented here avoids direct reliance on the notion of RC by reducing termination of system F to termination of bar recursion, that derives from an instance of Zorn's lemma.

In 1962, Spector used bar recursion [Spe62] to extend Gödel's Dialectica interpretation of arithmetic into an interpretation of analysis, showing that bar recursion interprets the axiom scheme of comprehension. Variants of bar recursion have then been used to interpret the axioms of countable and dependent choice in a classical setting through Kreisel's modified realizability. Among these variants, modified bar recursion [BO05] relies on the continuity of one of its arguments to ensure termination, rather than on the explicit termination condition of original bar recursion. The variant that we use is the BBC functional [BBC98], that builds the family of realizers in an order that depends on the order of computation rather than on the usual order on natural numbers. While in [BBC98], the proof of correctness of the BBC functional relies on syntactic arguments, we use an adaptation of the semantic proof of [Ber] that relies on Zorn's lemma. Using this we are then able to interpret the axiom scheme of comprehension, which is the only ingredient required on top of first-order arithmetic in order to get a computational interpretation of second-order arithmetic and therefore of normalization of system F.

For any single term of system F there exists a proof in second-order arithmetic that it terminates. This mapping from terms of system F to proofs of second-order arithmetic is closely related to Reynolds' abstraction theorem [Rey83] which, as explained in [Wad07], relies on an embedding of system F into second-order arithmetic. We use our bar recursive interpretation of second-order arithmetic to extract the normal form of the system F term from its termination proof. Our technique is similar to Berger's work in the simply-typed case [Ber93]. It is closely related to normalization by evaluation, extended to system F in [AHS96, Abe08]. To avoid an encoding of λ-terms as natural numbers, we define a multi-sorted first-order logic with a sort for λ-terms with de Bruijn indices. To formalize the notion of reducibility candidates, our logic also has a sort for sets of λ-terms. Since these are first-order elements

*Research supported by the UK EPSRC grant EP/P004172/1.

of the logic, the instantiation of a set variable with an arbitrary formula is not directly possible as it would be in second-order arithmetic. We will however get back this possibility through our bar recursive interpretation of the axiom scheme of comprehension.

In a second step we fix the target programming language of the translation. This language, that we call system ΛT_{br}, is purely functional with a type for λ-terms, primitive recursion, and the BBC functional. System ΛT_{br} is in particular simply-typed and total. We also describe the sound and computationally adequate semantics of this language in the category of complete partial orders.

The last step is the definition of a realizability semantics for our logic. To each formula we associate a type of system ΛT_{br} and a set of realizers in the domain interpreting that type. Realizers are elements of the model rather than syntactic programs because the correctness of the BBC functional requires the existence of non-computable functions on discrete types which only exist in the model. Since we encode existential quantifications using universal ones and negation, we are able to interpret classical logic. On the other hand, the BBC functional interprets a variant of the axiom of countable choice, and its combination with our interpretation of classical logic provides a realizer of the axiom scheme of comprehension. Using this, we are then able to interpret the instantiation of set variables with arbitrary formulas. Finally, we associate for each term of system F a program that interprets the proof of its termination for weak head reduction and computes the normal form of the initial term of system F.

References

[Abe08] Andreas Abel. Weak beta-eta-Normalization and Normalization by Evaluation for System F. In *15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 497–511. Springer, 2008.

[AHS96] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-Free Normalisation for a Polymorphic System. In *11th IEEE Symposium on Logic in Computer Science*, pages 98–106. IEEE Computer Society, 1996.

[BBC98] Stefano Berardi, Marc Bezem, and Thierry Coquand. On the Computational Content of the Axiom of Choice. *Journal of Symbolic Logic*, 63(2):600–622, 1998.

[Ber] Ulrich Berger. The Berardi-Bezem-Coquand-functional in a domain-theoretic setting. http://www-compsci.swan.ac.uk/ csulrich/ftp/bbc.ps.gz.

[Ber93] Ulrich Berger. Program Extraction from Normalization Proofs. In *1st International Conference on Typed Lambda Calculi and Applications*, pages 91–106. Springer, 1993.

[BO05] Ulrich Berger and Paulo Oliva. Modified bar recursion and classical dependent choice. In *Logic Colloquium '01*, volume 20 of *Lecture Notes in Logic*, pages 89–107. Springer-Verlag, 2005.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–69. North-Holland, 1971.

[Rey74] John Reynolds. Towards a theory of type structure. In *Programming Symposium, Paris, April 9-11, 1974*, Lecture Notes in Computer Science, pages 408–423. Springer, 1974.

[Rey83] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[Spe62] Clifford Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In *Recursive Function Theory: Proceedings of Symposia in Pure Mathematics*, volume 5, pages 1–27. American Mathematical Society, 1962.

[Wad07] Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1-3):201–226, 2007.

The Steenrod squares in homotopy type theory

Guillaume Brunerie

Institute for Advanced Study, Princeton, NJ

A few years ago, a strong connection between type theory and homotopy theory, known as "homotopy type theory", was discovered, and it has then been used to reason about homotopy theory using type-theoretic intuition. Several examples of results of homotopy theory done in this style are given in chapter 8 of the reference book [6], in my PhD thesis [1], and in the articles [2, 3, 4, 5], among others.

The Steenrod squares are operations on cohomology with $\mathbb{Z}/2\mathbb{Z}$ coefficients, related to the commutativity "up-to-homotopy" of the cup product, and it is a very important tool in classical homotopy theory. It turns out that the notion of commutativity up-to-homotopy has a very natural type-theoretic formulation, which gives a definition of the Steenrod squares in homotopy type theory. This is what we will present here.

Commutativity up to homotopy. Given a map $f : A \times A \to B$, one says that f is *commutative in the naive sense* if for every $x, y : A$ one has an equality $p_{x,y} : f(x, y) =_B f(y, x)$ (an element of the identity type). This is a good definition if B is a set in the sense of homotopy type theory (i.e., if it satisfies uniqueness of identity proofs), but not if B is an arbitrary type. Indeed, $p_{x,y}$ and $p^{-1}_{y,x}$ are two elements of the identity type $f(x, y) =_B f(y, x)$, and they ought to be related. Asking for $p_{x,y}$ and $p^{-1}_{y,x}$ to be equal is enough when B is a 1-type, but it creates a new coherence problem one dimension higher, and so on. It might look like the classical problem of infinite coherences in homotopy type theory, but it turns out that this version has a simple and very intuitive solution:

Definition 1. A *commutativity structure* on a map $f : A \times A \to B$ is a family of maps

$$f_X : A^X \to B$$

for every 2-element type X (i.e. for every X which is merely equivalent to Bool), and an identification of f_{Bool} with f, for the natural identification of A^{Bool} with $A \times A$.

In other words, a commutativity structure on f explains how to apply f to "any 2" elements of A, where "any 2" means that the elements are parametrized by an arbitrary 2-element type. In particular, it follows that f is commutative in the naive sense because by the univalence axiom there is an equality $\mathsf{Bool} = \mathsf{Bool}$ swapping the two elements of Bool, which gives a homotopy between $\lambda x, y.f(x, y)$ and $\lambda x, y.f(y, x)$. Moreover, that equality is equal to its opposite, hence we get the equality between $p_{x,y}$ and $p^{-1}_{y,x}$ mentioned above, and so on.

The connection with homotopy theory is contained in the following theorem, due to Egbert Rijke and Ulrick Buchholtz ([7]):

Theorem 1. The type of all 2-element types is equivalent to the infinite projective space $\mathbb{R}P^\infty$.

In particular, given a map $f : A \times A \to B$ with a commutativity structure, we obtain the *extended diagonal map*

$$\Delta_f : \mathbb{R}P^\infty \times A \to B,$$
$$\Delta_f(X, x) := f_X(\lambda\_.x)$$

which describes the structure obtained by swapping both x's in $f(x, x)$, and all the higher coherences associated to it.

The Steenrod squares. The *cup product* is an operation on the cohomology groups of a type. It is characterized by a map

$$\smile\ : K_n \times K_m \to K_{n+m},$$

where the types $K_n := K(\mathbb{Z}/2\mathbb{Z}, n)$ are Eilenberg-MacLane spaces (as defined, for instance, in [4]). The map $\smile$ is defined by using the fact that $\|K_n \wedge K_m\|_{n+m}$ is equivalent to K_{n+m}, where the *smash product* $A \wedge A'$ is defined as the pushout

$$A \wedge A' := (A \times A') \sqcup^{A \sqcup^{\mathbf{1}} A'} \mathbf{1}$$

(intuitively, the type $A \wedge A'$ is the product of A and A' where we contract to a point a copy of A and a copy of A'). Taking n and m to be equal, we would like to construct a commutativity structure on the cup product.

The main ingredient is that it is possible to construct a type $A^{\wedge X}$, for any two-element type X, which corresponds intuitively to the smash product of A with itself, but where both copies of A are indexed by X. The definition is as follows:

$$A^{\wedge X} := (A^X) \sqcup^{(A \times X) \sqcup^X \mathbf{1}} \mathbf{1}.$$

One can easily check that this type is equivalent to $A \wedge A$ when X is Bool, and swaps both copies of A when we swap the two elements of Bool.

We can then prove that $\|K_n^{\wedge X}\|_{2n}$ is equivalent to K_{2n} for any X, using the fact that we are working modulo 2, and therefore we get a commutativity structure on the cup product whose extended diagonal map is a map

$$\Delta_{\smile} : \mathbb{R}P^\infty \times K_n \to K_{2n}.$$

Finally, we can study the cohomology of $\mathbb{R}P^\infty$ and show that it gives rise to maps

$$\widetilde{\mathrm{Sq}}^i : K_n \to K_{n+i}$$

and therefore to maps

$$\mathrm{Sq}^i : H^n(X, \mathbb{Z}/2\mathbb{Z}) \to H^{n+i}(X, \mathbb{Z}/2\mathbb{Z})$$

(called the *Steenrod squares*), and then we can prove that they satisfy the same properties as the Steenrod squares in classical homotopy theory.

References

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

[2] Kuen-Bang Hou, Eric Finster, Daniel R. Licata, and Peter LeFanu Lumsdaine. A mechanization of the Blakers-Massey connectivity theorem in homotopy type theory. *LICS*, 2016.

[3] Daniel R. Licata and Guillaume Brunerie. $\pi_n(S^n)$ in homotopy type theory. *Invited paper, CPP*, 2013.

[4] Daniel R. Licata and Eric Finster. Eilenberg–MacLane spaces in homotopy type theory. *LICS*, 2014.

[5] Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. *LICS*, 2013.

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

[7] Egbert Rijke and Ulrik Buchholtz. The real projective spaces in HoTT, 2016. Slides available at https://egbertrijke.files.wordpress.com/2016/09/lc2016.pdf.

Verifying Functional Reactive Programs with Side Effects

Manuel Bärenz[1] and Sebastian Seufert[2]

[1] University of Vienna, Austria
maths@manuelbaerenz.de
[2] Otto-Friedrich-University of Bamberg, Germany
sebastian.seufert@stud.uni-bamberg.de

Temporal Logic and Verification of Functional Reactive Programming In general, desirable properties of reactive programs are expressed in a suited temporal logic. Recently, Alan Jeffrey has shown that Linear Temporal Logic (LTL) can be embedded into Martin-Löf type theory, allowing to write functional reactive programs and expressing their properties as LTL-formulas in a natural way [1].

What is lacking is an equally suitable embedded domain-specific language (EDSL) to express *proofs* of these properties, general enough to comply a broader range of cases while still practicable to the programmer. When employing such an EDSL with support by interactive assistants, ideally program, properties and proofs could then be provided together and thus increase the modularity of the verified code.

Jeffrey's approach covers pure signal processing without side effects. It is a practical programming pattern to allow side effects in FRP, though. Knowledge of the environment time, which distinguishes signal processing from stream processing, can be regarded as a side effect as well. Consequently, stream processing with side effects as described in [2] is a viable approach to hybrid FRP, combining discrete and continuous paradigms.

The work cited above describes a framework for effectful stream processing implemented in Haskell. Side effects are encoded as *monads* there, but other functors are also conceivable. The kind of side effect, represented by the choice of a particular functor, is then a parameter in the type signature of the reactive program. This allows for reasoning about its behaviour to some extent, similar as the EDSL provided by Alan Jeffrey.

However, in the context of verifying effectful FRP, there are propositions Φ whose values depend on the context of a side effect, such that LTL may not be sufficiently expressive:

Safety	"After any possible side effect that can occur, Φ becomes true."
Liveness	"There is a side effect such that if it occurs, Φ becomes true."

These two aspects match the 'all paths' (A) and 'exists a path' (E) modalities from Computational Tree Logic (CTL) excellently.

Container modalities With side effects encoded as completely positive functors, *container extensions* provide the possibility to express these modalities. A container is a dependent pair S ▷ P of a type S, the "shapes", and a type family P, the "positions". Every container gives a functor, its *extension*:

```
⟦ S ▷ P ⟧ X = Σ[ s ∈ S ] (P s → X)
```

A value of type ⟦ S ▷ P ⟧ X is a side-effectful computation that produces a value of type X. The shape s plays the role of a command that is sent to the environment, while the type of positions P s encodes the possible response values that the environment can supply in order to yield a definite value.

Implementation Our library is formalised in Agda. There, containers are universe-polymorphic. A value in `Set` encodes a proposition, thus a value in `⟦ S ▷ P ⟧ Set` is an *effectful proposition*, i.e. a proposition with its validity depending on the environment. Consequently, container modalities can be defined which encode the statements that the proposition holds for any response (or position), or that there exists a response such that the proposition holds, respectively.

We implement effectful streams (FStreams) as a coinductive type:

```
FStream : Container → Set → Set
FStream (S ▷ P) X = ν Y . ⟦ S ▷ P ⟧ (X × Y)
```

Each time a value is retrieved from the stream, a side effect is executed.

Our library encompasses the usual utilities to work with streams, such as streams repeating (temporally) constant effectful values, application of functions to streams, a syntax for defining ultimately periodic streams from their prefixes, corecursion and bisimulation. All the elements of the EDSL can also be used to reason about the stream. Values of type `FStream (S ▷ P) Set` then correspond to CTL formulas.

We provide all CTL modalities, i.e. combinations of `A` and `E` with the temporal modalities `G` ("globally"), `F` ("future"), and others, such as "next" and "until". We supply a closely matched EDSL for constructing proofs for the CTL formulas, such that programs, properties and proofs become succinct and readable in our library as test cases demonstrate.

Example Consider the following program:

```
trafficLight : FStream (Reader Bool) Bool
trafficLight = ⟨ return true ▶ read ⟩ ▶···
```

This program encodes a traffic light which unconditionally outputs `true` (encoding "green") in the first tick, and asks for input from a `Reader` environment in the second tick and outputs it. After that, the stream repeats. (The input could be supplied by another stream, or in theory by a physical sensor.) In our DSL, ▶ is a stream constructor and ··· stands for repetition.

We will verify a *responsivity property*: At any moment, the traffic light could be green, given the appropriate input.

```
responsivity : EG (map (true ≡_) trafficLight)
responsivity = mapEG ⟨ refl ▶EG refl ⟩EG ▶EG···
```

Since we prove a "*globally*"-modality, the whole proof will be a stream of proofs for every tick, and can be expressed in a DSL approximating the stream DSL. First, we commute `map` past `EG`. We then prove the tautology `true ≡ true` for the first tick. In the second tick, we prove that the stream can indeed output `true`, and Agda automatically infers the appropriate input that the user needs to make to validate the proof. Finally, the proof repeats.

With little more effort, one can implement a second traffic light and verify, for example, a safety property (the two traffic lights are never green at the same time, under any side effect).

References

[1] Alan Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, pages 49–60. ACM, 2012.

[2] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell*, pages 33–44. ACM, 2016.

Notions of type formers[*]

Paolo Capriotti

University of Nottingham

Martin-Löf Type Theory (MLTT) exists in a variety of incarnations, with several slightly different combinations of *type formers*, that are hard to express in a single definition, and study uniformly. In particular, when dealing with MLTT from a semantic point of view, it is often not clear how to establish a correct and sufficiently general notion of *model* of type theory equipped with any choice of type formers. By "type formers" I mean here constructs such as Π-types, Σ-types, equality types, etc., which occur, in more or less the same form, in most presentations of type theory, as well as more "specific" ones, of which there are multiple variations. Examples of these specific type formers include inductive types (e.g. simple formulations like W-types, or more involved ones like higher inductive [5] or inductive-inductive types [3]), coinductive types, or universes.

Therefore, when studying type theory from a meta-theoretical point of view, one often encounters the problem that one is forced to fix a particular incarnation of type theory upfront, so that the corresponding notion of *model* is determined. However, most of the times, it is the case that many meta-theoretical results that one might wish to prove are true in a wide variety of setups, and for different variations of type formers. Unfortunately, expressing this idea precisely is not immediately possible, since a general notion of "type former" on which results could be parameterised does not actually exist in a formal sense. Among the difficulties that one encounters when trying to give such a general definition is the issue of "coherence", i.e. the problem of finding a precise formulation for the stability properties of a type former under substitution. The lack of a general notion of model has the unfortunate consequence of making many proofs unnecessarily specific; it is often intuitively apparent that certain meta-theoretical results do not depend, to a certain extent, on a particular choice of type formers, but making this kind of statement precise is usually quite a challenge, for the reasons above.

To solve this problem, I propose a language for describing type formers based on the idea of a "logical framework" [1, 4]. The basic idea is to use type theory itself, i.e. one specific incarnation of MLTT, fixed once and for all, as the language in which type formers can be expressed. The theory that I choose for this purpose contains the following type formers: Π-types, Σ-types, equality types, a unit type and a universe with no structure. By "universe with no structure" I simply mean a distinguished type $\mathcal{U}$, together with a family El over it. All the other type formers are *extensional*, i.e. they admit strict β and η computation rules. This implies, in particular, that equality has the reflection rule. Using this basic theory, which I call RF (for *rule framework*), I can now give a definition of an arbitrary *type former*: it is simply a context in RF, or, equivalently, a type in the empty context. The Σ-types of RF are modelled, in a category with families (CwF) [2], by fibred left adjoints to reindexing functors along display maps; similarly, Π-types are given by fibred right adjoints to the same functors, while existence of equality types is equivalent to that of a fibred left adjoint to substitution functors along diagonal morphisms $\Gamma.A \to \Gamma.A.A$. It is therefore quite straightforward to define a notion of *model* for the RF system, which I will call *RF category*.

Now let $\mathcal{C}$ be any (small) CwF. The corresponding category of presheaves $\widehat{\mathcal{C}}$ is an RF category. In fact, it is well known that all the extensional type formers of RF can be interpreted in any presheaf category, so all we need to specify is how to interpret the universe. The type $\mathcal{U}$ is

[*]This work is supported by USAF grant FA9550-16-1-0029.

interpreted as the presheaf Ty of types in $\mathcal{C}$, and El as the presheaf Tm of terms (regarded as a presheaf over Ty). Therefore, it follows that one can interpret any type former Φ as a presheaf $[\![\Phi]\!]$ in $\widehat{\mathcal{C}}$. A Φ-structure of $\mathcal{C}$ (i.e. an instance of the Φ type former for $\mathcal{C}$) is then defined to be a *global section* of $[\![\Phi]\!]$.

The power of this simple definition is that the specification of a type former in the language of RF does not contain the usual boilerplate that is normally necessary in a traditional type-theoretic presentation, i.e. the so called *congruence* rules, that specify coherence conditions for a type former with respect to substitutions. In fact, the naturality condition of the corresponding global section automatically enforces strict forms of such coherence conditions. For example, let us consider a very simple type former, resembling a (trivial) *modality operator*, denoted $\Box$. In a traditional presentation, we could express it with the following rules:

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma \vdash \Box A \text{ type}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \eta(a) : \Box A} \qquad \frac{\Gamma.(\Box A) \vdash P \qquad \Gamma(a : A) \vdash d : P(\eta(a))}{\Gamma.(\Box A) \vdash \mathsf{elim}^P(d) : P}.$$

The first rule can be directly translated into the language of RF as the context corresponding to the type $\mathcal{U} \to \mathcal{U}$. The interpretation of $\mathcal{U} \to \mathcal{U}$ in $\widehat{\mathcal{C}}$ is the presheaf exponential $[\mathsf{Ty}, \mathsf{Ty}]$, and a global section of it is simply a natural transformation $\mathsf{Ty} \to \mathsf{Ty}$, which can be unfolded as an assignment $A \mapsto \Box A$, satisfying a condition of (strict) stability under substitution. The whole type former can be expressed in RF as the following type:

$$\begin{aligned}\Pi(A : \mathcal{U}), \Sigma(A' : \mathcal{U}), \Sigma(\eta : \mathsf{El}[A] \to \mathsf{El}[A']),& \\ (\Pi(P : \mathsf{El}[A'] \to \mathcal{U}), \Pi(d : \Pi(a : A), P(\eta(a))), \Pi(a' : A'), P(a')),&\end{aligned} \tag{1}$$

where we factored out the premise $A : \mathcal{U}$, which appears in each of the three rules above, into a single outermost Π-type. Note that the generic context Γ, appearing in every single judgement of the rule-based presentation, is not explicitly mentioned in the corresponding RF type. The dependency of the rules on a generic context is automatically reintroduced by the interpretation of (1) in presheaves. Note that, when in a rule the context Γ appears extended, this is expressed in RF by using a $\mathcal{U}$-valued function type, like for example for the type of η in (1). We could also add a computation rule using the equality type of RF.

All the usually employed type formers can be encoded in a similar way, including arbitrarily nested (but finite) systems of universes, and special cases of higher inductive types. For any such type former encoded in the RF system, we therefore get a corresponding notion of model, and using a technique based on the idea of *logical relations*, we can get an associated notion of *morphism*. Unfortunately, at this level of generality it is not true that models of any given type former form a category, since composition of morphisms cannot be shown to be associative. I am currently exploring ways to refine the language of RF so that structures corresponding to RF types always form a category, possibly monadic over the category of CwFs.

References

[1] Nicolaas Govert de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on automatic demonstration*, pages 29–61. Springer, 1970.

[2] Peter Dybjer. Internal type theory. In *Types for Proofs and Programs*, pages 120–134. Springer, 1995.

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

[4] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

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

Automatically Constructing a Type System from the Small-Step Semantics

Ștefan Ciobâcă and Vlad Andrei Tudose*

Faculty of Computer Science,
"Alexandru Ioan Cuza" University,
Iași, Romania
stefan.ciobaca@info.uaic.ro

Abstract

We describe preliminary work suggesting that most typing rules for a programming language can be obtained automatically from its operational semantics. Instead of going the usual way, to first define the semantics and the type system, and then show progress and preservation, we start from the semantics and construct a type system that satisfies progress and preservation by construction. We have tested our approach on simple lambda calculi and we have constructed a Haskell prototype that implements our algorithm.

1 Inferring Typing Rules

Usually, type soundness proofs for programming languages are quite straightforward. This is not too surprising, since most typing rules are quite similar to the operational semantics rules, when looking at them from a meta-syntactic point of view. We conjecture that most typing rules for a language can be generated automatically from the operational semantics. Type soundness should then follow immediately, by construction.

Imagine a programming language with the following syntax:

```
<t> ::= true | false | if <t> then <t> else <t> | 0 |
succ <t> | pred <t> | isZero <t>
<nv> ::= 0 | succ <nv>
<bv> ::= true | false
<v> ::= <nv> | <bv>
```

and the folllowing associated operational semantics:

$$\texttt{if true then } \langle t_2\rangle \texttt{ else } \langle t_3\rangle \to \langle t_2\rangle \qquad \texttt{if false then } \langle t_2\rangle \texttt{ else } \langle t_3\rangle \to \langle t_3\rangle$$

$$\frac{\langle t_1\rangle \to \langle t_1'\rangle}{\texttt{if } \langle t_1\rangle \texttt{ then } \langle t_2\rangle \texttt{ else } \langle t_3\rangle \to \texttt{if } \langle t_1'\rangle \texttt{ then } \langle t_2\rangle \texttt{ else } \langle t_3\rangle} \qquad \frac{\langle t_1\rangle \to \langle t_1'\rangle}{\texttt{succ } \langle t_1\rangle \to \texttt{succ } \langle t_1'\rangle}$$

$$\texttt{pred 0} \to \texttt{0} \qquad \texttt{pred (succ } \langle t_1\rangle\texttt{)} \to \langle t_1\rangle \qquad \frac{\langle t_1\rangle \to \langle t_1'\rangle}{\texttt{pred } \langle t_1\rangle \to \texttt{pred } \langle t_1'\rangle} \qquad \texttt{isZero 0} \to \texttt{true}$$

$$\texttt{isZero (succ } \langle t_1\rangle\texttt{)} \to \texttt{false} \qquad \frac{\langle t_1\rangle \to \langle t_1'\rangle}{\texttt{isZero } \langle t_1\rangle \to \texttt{isZero } \langle t_1'\rangle}$$

*The work reported in this paper was performed while the second author was a Master's student at the Faculty of Computer Science.

What could be the typing rules for the language? Next, we will assume that all typing rules have the following shape, for all operators `op` of the language:

$$\frac{\langle t_1\rangle : \langle T_1\rangle \qquad \langle t_2\rangle : \langle T_2\rangle \qquad \ldots \qquad \langle t_n\rangle : \langle T_n\rangle}{\texttt{op}\ \langle t_1\rangle\ \langle t_2\rangle\ \ldots\ \langle t_n\rangle : \langle T\rangle} \quad (\mathit{Name})$$

Assume that we know the typing rules for `true` and `0` are:

$$\frac{}{\texttt{true : Bool}} \qquad \frac{}{\texttt{0 : Nat}}$$

and suppose that our typing system has the preservation property. When we look at the rule for `isZero`:

`isZero 0` $\to$ `true`

it is immediate that `isZero` 0 must also be of type `Bool` (otherwise preservation would not hold). Furthermore, by our assumption on the set of operators `op` of the language and since `0 : Nat`, the `isZero` operator must satisfy the following typing rule:

$$\frac{\langle t_1\rangle : \texttt{Nat}}{\texttt{isZero}\ \langle t_1\rangle : \texttt{Bool}}$$

By reasoning similar to the above, we can infer the expected typing rules for all other language operators, as presented in [1]. Any well-typed term in the resulting type system does not get stuck. We have implemented and shown that our algorithm for transforming the operational semantics into typing rules is sound, in the sense that the resulting typing system has progress and preservation. Furthermore, we have extended our approach to handling more complex typing rules, which have the following shape:

$$\frac{\Gamma_1 \vdash \langle t_1\rangle : \langle T_1\rangle \qquad \Gamma_2 \vdash \langle t_2\rangle : \langle T_2\rangle \qquad \ldots \qquad \Gamma_n \vdash \langle t_n\rangle : \langle T_n\rangle}{\Gamma \vdash \texttt{op}\ \langle t_1\rangle\ \langle t_2\rangle\ \ldots\ \langle t_n\rangle : \langle T\rangle} \quad (\mathit{Name})$$

This allows us to infer the typing rules for simple lambda calculi presented in Curry style.

Our prototype implementation, written in Haskell, can be accessed at `http://profs.info.uaic.ro/~stefan.ciobaca/types2017`. The program takes as input the files `syntax.txt` (describing the syntax of the language in a BNF-like format), `eval_rules.txt` (describing the small-step SOS of the language), `type_syntax.txt` (describing the syntax of the types of the language) and `typing.txt` (describing the basic typing rules, such as `0 : Nat`) and outputs the rest of the typing rules.

2 Discussion

This work could simplify type system design for programming languages, as there would be no need to prove progress and preservation. Our approach cannot currently handle sub-typing, polymorphism or exception handling and we leave these features for further work. The final goal is to start from a formal semantics, for example reified from a Coq development, assume progress and preservation, and automatically derive a mechanically proven sound-by-construction typing system.

References

[1] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

A Model of Type Theory in Stacks[*]

Thierry Coquand[1], Bassel Mannaa[2], and Fabian Ruch[3]

[1] University of Gothenburg, Sweden `thierry.coquand@cse.gu.se`
[2] IT University of Copenhagen, Denmark `basm@itu.dk`
[3] University of Gothenburg, Sweden `fabian.ruch@cse.gu.se`

We give a model of dependent type theory with one univalent universe and propositional truncation interpreting a type as a *stack*, generalizing the groupoid model of type theory. As an application, we show that countable choice and Markov's principle cannot be proved in dependent type theory with one univalent universe and propositional truncation. We work in constructive metatheory.

For simple type theory such independence results can be obtained by using *sheaf semantics*, respectively over Cantor space (for Markov's principle) and open unit interval $(0, 1)$ (for countable choice). There are however problems with extending sheaf semantics to universes [5, 6]. In order to address these issues we use a suitable formulation of *stack semantics*, which, roughly speaking, replaces *sets* by *groupoids*. The notion of stack was introduced in algebraic geometry [3] precisely in order to solve the same problems that one encounters when trying to extend sheaf semantics to type-theoretic universes. The compatibility condition for gluing local data is now formulated in terms of isomorphisms instead of strict equalities. In this sense, our model can also be seen as an extension of the groupoid model of type theory [4].

The argument should generalize to an ∞-stack version of the cubical set model [1]. The coherence condition on descent data will be infinitary in general, but it will become finitary when we restrict the homotopy level (and empty in particular in the case of propositions).

Stacks over a topological space. Let X be a topological space, U an open set and (U_i) a covering of U. We denote by U_{ij} the intersection of U_i and U_j, and by U_{ijk} the intersection of U_i, U_j and U_k.

A *prestack* over X is a presheaf F of groupoids over X such that given $u, u' \in F(U)$ and a compatible family of paths $\omega_i : u|U_i \cong u'|U_i$ in $F(U_i)$, i.e. $\omega_i|U_{ij} = \omega_j|U_{ij}$, there is a unique path $\omega : u \cong u'$ in $F(U)$ with $\omega|U_i = \omega_i$.

A *descent datum* for F is given by a covering (U_i) of U and a family of objects $u_i \in F(U_i)$ with paths $\varphi_{ij} : u_i|U_{ij} \cong u_j|U_{ij}$ satisfying the *cocycle* condition

$$\varphi_{ii} = 1_{u_i} \qquad \varphi_{ij}|U_{ijk} \cdot \varphi_{jk}|U_{ijk} = \varphi_{ik}|U_{ijk}$$

A *stack* over X is a prestack equipped with a gluing operation **glue** mapping a descent datum (u_i, φ_{ij}) to an object $u \in F(U)$ and paths $\varphi_i : u|U_i \cong u_i$ satisfying

$$\varphi_i|U_{ij} \cdot \varphi_{ij} = \varphi_j|U_{ij} \qquad \mathbf{glue}(u_i, \varphi_{ij})|V = \mathbf{glue}(u_i|V \cap U_i, \varphi_{ij}|V \cap U_{ij}) = (u|V, \varphi_i|V \cap U_i)$$

A prime example of a stack whose presheaf of objects is not a sheaf is the universe of sheaves: If we define $F(U)$ to be the collection of small sheaves over U then there is a natural restriction operation $F(U) \to F(V)$ for $V \subseteq U$, and one can check that the gluing of a compatible family of elements is not unique up to strict equality in general (but it is unique up to isomorphism).

Interpretation of type theory in stacks. Starting from the above notion of stack, it is possible to design a model of dependent type theory which extends both the sheaf model of simple type theory and the groupoid model [4].

As in the groupoid model, in this model we have a univalent universe U (corresponding to the stack of small sheaves) and propositional truncation $\|\cdot\|$. We also have a notion of small discrete type and a universal family $\mathsf{El}\, X$ $(X : \mathsf{U})$ of small discrete types over the universe, i.e. for every small discrete type A there exists a unique section $|A| : \mathsf{U}$ such that $\mathsf{El}\ |A| = A$.

[*]This abstract is based on the paper [2] accepted for LICS'17.

The unit interval and countable choice. We express countable choice in type theory by:

$$\mathsf{CC} = \Pi(A : \mathsf{N} \to \mathsf{U})(\Pi(n : \mathsf{N}) \,\|\mathsf{El}\,(A\,n)\|) \to \|\Pi(n : \mathsf{N})\mathsf{El}\,(A\,n)\|$$

We write $U, V, W, \ldots$ for nonempty open rational intervals included in the open unit interval $(0, 1)$. We consider the interpretation of type theory in stacks over this space.

We let $|\mathsf{N}|$ be the constant sheaf where each $|\mathsf{N}|(V)$ is the set $\mathbb{N}$ of natural numbers and $\mathsf{N} = \mathsf{El}\,|\mathsf{N}|$.

Define $A : \mathsf{N} \to \mathsf{U}$ where $A\,n$ is a subsheaf of the (small) constant sheaf $|\mathsf{Q}|(V) = \mathbb{Q}$ of rational numbers.

$$(A\,n)(V) = \left\{ r \in \mathbb{Q} \;\middle|\; \forall (x \in V)\ |x - r| < \frac{1}{n+1} \right\}$$

Proposition 1. *In this model*

1. *the type* $\Pi(n : \mathsf{N})\,\|\mathsf{El}\,(A\,n)\|$ *is inhabited*
2. *the type* $\Pi(n : \mathsf{N})\mathsf{El}\,(A\,n)$*, and hence also the type* $\|\Pi(n : \mathsf{N})\mathsf{El}\,(A\,n)\|$*, is empty*

Corollary 1. *In this model, the principle of countable choice* CC *does not hold. Consequently, one cannot show countable choice in type theory with one univalent universe and propositional truncation.*

Cantor space and Markov's principle. We express Markov's principle in type theory by:

$$\mathsf{MP} := \Pi(h : \mathsf{N} \to \mathsf{N}_2)(\neg\neg(\Sigma(x : \mathsf{N})\mathsf{El}\,\mathsf{isZero}\,(h\,x)) \to \Sigma(x : \mathsf{N})\mathsf{El}\,\mathsf{isZero}\,(h\,x))$$

where $\mathsf{isZero} : \mathsf{N}_2 \to \mathsf{U}$ is defined by $\mathsf{isZero} := \lambda y.\mathsf{rec}_2\ |\mathsf{N}_1|\ |\mathsf{N}_0|\ y$ and the type $\neg A$ by $A \to \mathsf{N}_0$.

We assume that the basic opens are nonzero elements $e, e', \ldots$ of a boolean algebra with decidable equality. We consider only coverings of e given by a finite partition e_i, $i \in I$, of e, that is a *finite set* of disjoint elements $e_i \leqslant e$ such that $e = \bigvee_{i \in I} e_i$.

Take a countably infinite set of variables $p_0, p_1, \ldots$. Consider the free boolean algebra generated by the atomic formulae p_n. We write $p_n = 0$ for $\neg p_n$ and $p_n = 1$ for p_n. An object e in this algebra represents then a compact open in Cantor space $\{0, 1\}^{\mathbb{N}}$, where a conjunctive formula $\bigwedge_{i \in I} p_i = b_i$ represents the set of sequences in $\{0, 1\}^{\mathbb{N}}$ having value b_i at index i. A formula e in the algebra is then a finite disjunction of these.

We consider the model of type theory in stacks over this algebra.

Proposition 2. *In this model*

1. $\neg\neg(\Sigma(x : \mathsf{N})\,\mathsf{El}\,\mathsf{isZero}\,(\mathsf{f}\,x))$ *is inhabited.*
2. $\Sigma(x : \mathsf{N})\,\mathsf{El}\,\mathsf{isZero}\,(\mathsf{f}\,x)$ *is not inhabited.*

Corollary 2. *In this model Markov's principle does not hold. Consequently, one cannot show Markov's principle in type theory with one univalent universe.*

References

[1] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[2] T. Coquand, B. Mannaa, and F. Ruch. Stack semantics of type theory. *CoRR*, abs/1701.02571, 2017.

[3] A. Grothendieck and J. Dieudonné. Éléments de géométrie algébrique. I. Le langage des schémas. *Institut des Hautes Études Scientifiques. Publications Mathématiques*, (4):228, 1960.

[4] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

[5] M. Hofmann and T. Streicher. Lifting Grothendieck universes. December 2014.

[6] C. Xu and M. Escardó. Universes in sheaf models. February 2016.

The Boolean Pythagorean Triples Problem in Coq [*]

Luís Cruz-Filipe[1], Joao Marques-Silva[2], and Peter Schneider-Kamp[1]

[1] Department of Mathematics and Computer Science, University of Southern Denmark
`{lcf,petersk}@imada.sdu.dk`
[2] LaSIGE, Faculty of Science, University of Lisbon, Portugal
`jpms@ciencias.ulisboa.pt`

The Boolean Pythagorean Triples problem asks the following question: is it possible to partition the natural numbers into two sets such that no set contains a Pythagorean triple (three numbers a, b and c with $a^2 + b^2 = c^2$)? This question was answered in 2016, when Heule, Kullmann and Marek [4] showed that it is already impossible to partition the set $\{1, \ldots, 7825\}$ into two sets such that none of them contains a Pythagorean triple. This proof was done by means of an encoding of this finite version of the problem into propositional logic (already used in [1]), which was then simplified and solved using the cube-and-conquer method [5].

The strategy of the proof is summarized in Figure 1. The propositional formula obtained by encoding the problem was first simplified using blocked clause elimination and symmetry breaking. Afterwards, the problem was divided into one million *cubes*: a set of partial assignments that cover the whole space of possible valuations. Then, it was shown that (1) the conjunction of the simplified formula with any cube is unsatisfiable, and (2) the negation of the disjunction of all the cubes is unsatisfiable. As a consequence, the simplified formula (and therefore also the original formula) is unsatisfiable.

This proof was formally verified using Coq, as described in [2, 3]. In this extended abstract, we summarize this process in a unified way.

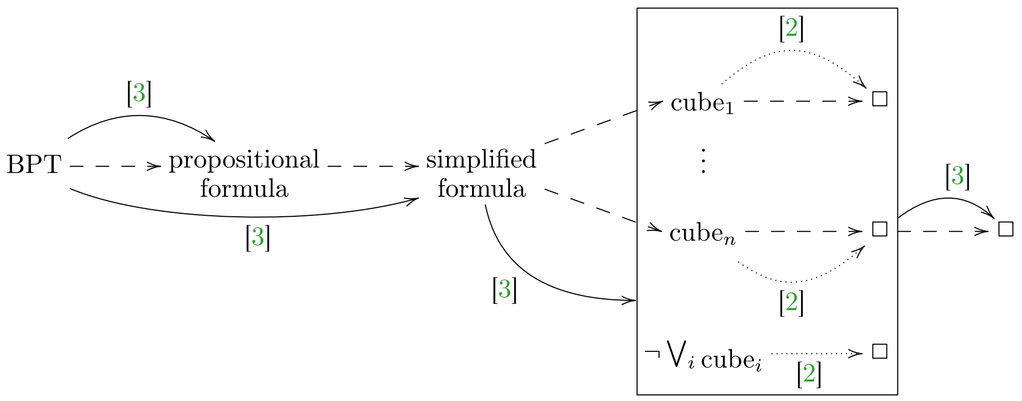


Figure 1: The original proof and the different verification steps. The dashed arrows denote the steps in the original proof [4]: a first propositional formula was generated by a C program, and subsequently simplified, divided and solved by SAT solvers. The dotted arrows denote proofs of unsatisfiability obtained by a SAT solver that were verified by a certified checker extracted from a Coq formalization [2]. The solid arrows denote the contributions of [3]: the generation, in Coq, of propositional formulas that are proved to represent the original mathematical problem, directly and after simplification; the formal specification of the simple reasoning behind cube-and-conquer; and the generation of the formulas that are given as input to cube-and-conquer.

*This work was partially supported by the Danish Council for Independent Research, Natural Sciences, grant DFF-1323-00247.

The first step was to formalize the Boolean Pythagorean Triples problem in Coq and relate it to the propositional encoding. This amounted to stating the mathematical problem in Coq and defining a family of propositional formulas (parametrized on a natural number n) directly corresponding to the formulas used in [4] as the starting point. We formally proved that unsatisfiability of any of these formulas implies that the given mathematical problem does not have a solution.

The second step was to formalize the simplification procedure. This is trivial, as the techniques applied in [4] only remove clauses, trivially preserving satisfiability. However, the authors of [4] make a stronger claim – namely, that their criterion for removing clauses also guarantees preservation of unsatisfiability. The mathematical argument, which we formalized, is as follows: let L be a list of triples and $(a, b, c) \in L$ be a triple containing a number that does not occur in any other triple in L. If there is a coloring C of the natural numbers such that no triple in $L \setminus \{(a, b, c)\}$ is monochromatic and e.g. a does not occur in any other triple in L, then we can extend C by changing the color of a, if necessary.

The next step was to formalize soundess of cube-and-conquer [5]. The idea behind this methodology is simple: instead of looking for a satisfying assignment for a particular formula, consider its conjunctions with different sets of literals (the cubes) such that every possible assignment satisfies one of the possible cubes. For example, if φ is a formula on two variables x and y, the cubes could be $\{x\}$, $\{\bar{x}, y\}$ and $\{\bar{x}, \bar{y}\}$, and instead of φ we consider the three formulas $\varphi \wedge x$, $\varphi \wedge \bar{x} \wedge y$ and $\varphi \wedge \bar{x} \wedge \bar{y}$. Furthermore, to ensure that every assignment satisfies one of the cubes, we need to check that the formula $\bar{x} \wedge (x \vee \bar{y}) \wedge (x \vee y)$ is unsatisfiable. We formalized this argument for the general case, given a formula and a list of cubes.

The last step was to check unsatisfiability of all the formulas generated by cube-and-conquer. This was done by developing a general-purpose verifier of unsatisfiability proofs based on reverse unit propagation [2]. This verifier, also formalized in Coq, checks that a given formula entails the empty clause by following a list of steps given as oracle. This list of steps is produced from a trace of an untrusted SAT solver, and is essentially a list of pairs (ψ, ℓ) where ψ is a clause to be added and ℓ is a list of indices of already known clauses that entail ψ by reverse unit propagation. (For efficiency, we also include deletion of clauses that are no longer relevant.)

Due to the huge size of the traces involved (over 200 TB), it is infeasible to perform the whole verification process inside Coq; thus, we use program extraction to obtain code that is correct by construction, and we rely on metalevel reasoning to chain the different steps in the process. However, we reduce this dependency to checking that the same arguments are provided to different functions. In principle, given enough resources, the whole verification could have been done inside Coq.

References

[1] J. Cooper and R. Overstreet. Coloring so that no pythagorean triple is monochromatic. *CoRR*, abs/1505.02222, 2015.

[2] L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp. Efficient certified resolution proof checking. In *TACAS*, volume 10205 of *LNCS*, pages 118–135. Springer, 2017.

[3] L. Cruz-Filipe and P. Schneider-Kamp. Formally verifying the boolean pythagorean triples conjecture. In *LPAR 21*. EasyChair, accepted for publication.

[4] M. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *SAT*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016.

[5] M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC 2011*, volume 7261 of *LNCS*, pages 50–65. Springer, 2012.

Towards practical out-of-order unification

Nils Anders Danielsson[*] and Víctor López Juan

University of Gothenburg and Chalmers University of Technology

Some implementations of type theories have a mechanism for implicit arguments, that allows users to omit some code from their programs and proofs. This kind of feature can be useful. However, at least in the case of Agda [8] it can also be slow, and sometimes one might wish that more implicit arguments could be instantiated. We are chipping away at these problems, and this note describes some preliminary results.

We have constructed an example that is too demanding for the current implementation of Agda. This example involves the definition of a dependently typed object language inside Agda, and then a definition of "setoid" inside this language. The definition of the object language only allows well-typed terms to be constructed, following McBride [6]. The definition of "setoid" in this language involves a large number of implicit arguments that the type-checker has to infer.

Our prototype, a variant of Tog [5], improves upon Agda by managing to type-check the example, and doing so in reasonable time and space.[1] The main techniques used to achieve this result are heterogeneous unification in the style of Gundry and McBride [4, 3], and hash consing inspired by Shao et al. [9] and Filliâtre and Conchon [2]. Note, however, that the prototype excludes several features of Agda, including termination checking, and that this should be kept in mind when benchmark data is analysed.

Homogeneous vs. heterogeneous unification Dependent type-checking can be reduced to solving a set of *general constraints* of the form $\Gamma \vdash t : A \approx u : B$ [5]. Such a constraint is well-formed if in context Γ term t has type A and term u has type B. Implicit arguments in the program syntax are represented by typed metavariables in the constraint syntax. When a metavariable α is instantiated by a term t (which we require to be closed), any occurrence of α becomes definitionally equal to t. A constraint is solved by instantiating each metavariable in such a way that the two sides of the constraint become definitionally equal.

In a **homogeneous** unification approach, the two sides of a constraint have the same type. Mazzoli and Abel [5] split each general constraint $\Gamma \vdash t : A \approx u : B$ into two homogeneous *unifier constraints*, $C_1 \equiv \Gamma \Vdash A \approx B : \mathrm{Set}$ and $C_2 \equiv \Gamma \Vdash t \approx u : A$, where C_2 is only meaningful once C_1 has been solved. If the user program is ill-typed and C_2 is solved before C_1, then one can end up in a situation where the type-checker treats ill-typed terms as being well-typed [8]. Tog in homogeneous mode (and, to a lesser degree, Agda) will not tackle C_2 until C_1 is solved. This approach is too strict for our example, because it prevents unification under binders (λ and Π) until the types of the bound variables have been unified.

By contrast, in the **heterogeneous** approach due to Gundry and McBride [4], unifier constraints can have two different types: $C_1 \equiv \Gamma \Vdash A : \mathrm{Set} \approx B : \mathrm{Set}$, $C_2 \equiv \Gamma \Vdash t : A \approx u : B$. The constraint C_2 can now be tackled before C_1. In the case where both sides of a constraint are headed by a binder (e.g. $\Gamma \Vdash \lambda y.t : \Pi(x : A_1)B_1 \approx \lambda y.u : \Pi(x : A_2)B_2$), the constraint is simplified by introducing a twin variable $\hat{x}$ of dual type $A_1 \ddagger A_2$, defined so that $\acute{x} : A_1$ and $\grave{x} : A_2$. The new constraint becomes $\Gamma, \hat{x} : A_1 \ddagger A_2 \Vdash t[\acute{x}/y] : B_1 \approx u[\grave{x}/y] : B_2$.

If a constraint reaches an irreducible form, say $\Gamma \Vdash \alpha\, u_1 \ldots u_n : A \approx t : B$, and the equation is in the pattern fragment [7, 1], then there will be a unique solution, here

[*]Supported by a grant from the Swedish Research Council (621-2013-4879).

[1]However, note that we have not proved that Tog is correctly implemented, so we give no guarantee that programs that are accepted by Tog are actually well-typed.

$\alpha := \lambda u_1 \ldots u_n.t$. However, before instantiating α, the unifier must perform a compatibility check: $\Gamma \Vdash A : \mathrm{Set} \approx B : \mathrm{Set}$ must hold, and ditto for the two types of any twin variable $\hat{x}$ whose two projections both occur in the constraint. Thus it may seem as if the heterogeneous approach is no more powerful than the homogeneous approach. However, as the unifier works on C_2, intermediate constraints may be generated which, when solved, lead to the instantiation of metavariables. This additional information could be just what is needed to solve C_1.

Our implementation We have extended the prototype Tog [5] with a heterogeneous unifier based on Gundry and McBride's algorithm [4]. If a naive implementation of heterogeneous unification is used, then there is a risk that a significant amount of redundant work is performed when the types of left-hand and right-hand sides are similar. We use hash consing, and assign a unique identifier to each term. We can then memoize term-traversing operations (including normalization, substitution, computation of metavariables and free variables in terms, pruning of redundant metavariable arguments, η-expansion, and twin variable removal).

Preliminary results We have compared our heterogeneous unifier, an updated version of Tog's homogeneous unifier, and Agda.[2] Tog with the heterogeneous unifier type-checks the setoid example mentioned above in 131 s with hash consing and memoization (HC&M) disabled, and in 22 s with HC&M enabled. Tog with the homogeneous unifier deems the example unsolvable in 8 s with HC&M enabled, and Agda runs out of memory after more than 5 h. However, in other cases Tog (with the heterogeneous unifier and HC&M enabled) runs slower than Agda. It is an open question whether the use of HC&M can sometimes lead to excessive memory usage due to large memo tables.

We note that this case study provides one data point in favour of Gundry's belief that heterogeneous unification with twin variables [4] "handles a sufficiently broad class of problems to be useful for elaboration of a dependently typed language" [3, §4.3.4].

References

[1] Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In *TLCA 2011*. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-21691-6_5.

[2] Jean-Christophe Filliâtre and Sylvain Conchon. Type safe modular hash-consing. In *ML'06*, 2006. doi:10.1145/1159876.1159880.

[3] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[4] Adam Gundry and Conor McBride. A tutorial implementation of dynamic pattern unification. Unpublished, 2012. URL `http://adam.gundry.co.uk/pub/pattern-unify/`.

[5] Francesco Mazzoli and Andreas Abel. Type checking through unification. Preprint arXiv:1609.09709v1 [cs.PL], 2016.

[6] Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *WGP'10*, 2010. doi:10.1145/1863495.1863497.

[7] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 1992. doi:10.1016/0747-7171(92)90011-R.

[8] Ulf Norell and Catarina Coquand. Type checking in the presence of meta-variables. Unpublished, 2007. URL `http://www.cse.chalmers.se/~ulfn/papers/meta-variables.html`.

[9] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *ICFP '98*, 1998. doi:10.1145/289423.289460.

[2]Tog: `https://lopezjuan.com/project/togt`, commit `1c7c2ce`. Agda: `https://github.com/agda/agda`, tag `v2.5.2`. Environment: Fedora 25 x64, Intel 2630QM, 16 GB RAM, GHC 8.0.1, up to 2 GB runtime heap.

Models and termination of proof reduction in the $\lambda\Pi$-calculus modulo theory

Gilles Dowek

Inria and École normale supérieure de Paris-Saclay
gilles.dowek@ens-paris-saclay.fr

Models and termination

In Predicate logic, a model is defined by a domain $\mathcal{M}$, a set $\mathcal{B}$ of truth values, and an interpretation function, parametrized by a valuation ϕ, mapping each term t to an element $[\![t]\!]_\phi$ of $\mathcal{M}$, and each proposition A to an element $[\![A]\!]_\phi$ of $\mathcal{B}$.

Predicate logic can be extended to Deduction modulo theory, where a congruence on propositions is added. Proofs of a proposition A are then considered to also be proofs of any proposition congruent to A. In Deduction modulo theory, like in Predicate logic, a model is defined by a domain $\mathcal{M}$, a set $\mathcal{B}$ of truth values, and an interpretation function.

Usually, the set $\mathcal{B}$ is the two-element set $\{0, 1\}$, but the notion of model can be extended to a notion of many-valued model, where $\mathcal{B}$ is an arbitrary Boolean algebra, a Heyting algebra, a pre-Boolean algebra, or a pre-Heyting algebra. Boolean algebras permit to introduce intermediate truth values for propositions that are neither provable nor disprovable, Heyting algebras to construct models of constructive logic, and pre-Boolean and pre-Heyting algebras, where the order relation $\leq$ is replaced by a pre-order relation, to distinguish a notion of weak equivalence: $[\![A]\!]_\phi \leq [\![B]\!]_\phi$ and $[\![B]\!]_\phi \leq [\![A]\!]_\phi$, for all ϕ, from a notion of strong equivalence: $[\![A]\!]_\phi = [\![B]\!]_\phi$, for all ϕ. In Deduction modulo theory, the first corresponds to the provability of $A \Leftrightarrow B$ and the second to the congruence.

In a model valued in a Boolean algebra, a Heyting algebra, a pre-Boolean algebra, or a pre-Heyting algebra, a proposition A is said to be valid when it is weakly equivalent to the proposition $\top$, that is when, for all ϕ, $[\![A]\!]_\phi \geq \tilde{\top}$. A congruence $\equiv$ defined on propositions is said to be valid when, for all A and B such that $A \equiv B$, A and B are strongly equivalent, that is, for all ϕ, $[\![A]\!]_\phi = [\![B]\!]_\phi$.

Proof reduction terminates in Deduction modulo a theory defined by a congruence $\equiv$, when this theory has a model valued in the pre-Heyting algebra of reducibility candidates. As a consequence, proof reduction terminates if the theory is super-consistent, that is if, for all pre-Heyting algebras $\mathcal{B}$, it has a model valued in $\mathcal{B}$. This theorem separates the semantic and the syntactic aspects that are often mixed in the usual proofs of termination of proof reduction. The semantic aspect is in the proof of super-consistency of the considered theory and the syntactic in the universal proof that super-consistency implies termination of proof reduction.

The $\lambda\Pi$-calculus modulo theory

In Predicate logic and in Deduction modulo theory, terms, propositions, and proofs belong to three distinct languages. But, it is more thrifty to consider a single language, such as the $\lambda\Pi$-calculus modulo theory, and express terms, propositions, and proofs, in this language. Like the $\lambda\Pi$-calculus, the $\lambda\Pi$-calculus modulo theory is a λ-calculus with dependent types, but, like in Deduction modulo theory, its conversion rule is extended to an arbitrary congruence.

From pre-Heyting algebras to Π-algebras

The first goal of this talk is to extend the notion of pre-Heyting algebra to a notion of Π-algebra, adapted to the $\lambda\Pi$-calculus modulo theory. In Predicate logic and in Deduction modulo theory, the propositions are built from atomic propositions with the connectors and quantifiers $\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, and $\exists$. Accordingly, the operations of a pre-Heyting algebra are $\tilde{\top}$, $\tilde{\bot}$, $\tilde{\wedge}$, $\tilde{\vee}$, $\tilde{\Rightarrow}$, $\tilde{\forall}$, and $\tilde{\exists}$. In the $\lambda\Pi$-calculus and in the $\lambda\Pi$-calculus modulo theory, the only connector is Π. Thus, a Π-algebra mainly has an operation $\tilde{\Pi}$. As expected, its properties are a mixture of the properties of the implication and of the universal quantifier of the pre-Heyting algebras.

Layered models

The second goal of this talk is to extend the usual notion of model to the $\lambda\Pi$-calculus modulo theory.

Extending the notion of model to many-sorted predicate logic requires to consider not just one domain $\mathcal{M}$, but a family of domains $\mathcal{M}_s$ indexed by the sorts. For instance, in a model of Simple type theory, the family of domains is indexed by simple types. In the $\lambda\Pi$-calculus modulo theory, the sorts also are just terms of the calculus. Thus, we shall define a model of the $\lambda\Pi$-calculus modulo theory by a family of domains $(\mathcal{M}_t)_t$ indexed by the terms of the calculus and a function $[\![.]\!]$ mapping each term t of type A and valuation ϕ to an element $[\![t]\!]_\phi$ of $\mathcal{M}_A$.

The functions $\mathcal{M}$ and $[\![.]\!]$ are similar, in the sense that both their domains is the set of terms of the calculus. The goal of the model construction is to define the function $[\![.]\!]$ and the function $\mathcal{M}$ can be seen as a tool helping to define this function. For instance, if f is a constant of type $A \to A$, where A is a term of type $Type$, and we have the rule $f(x) \longrightarrow x$, we want to define the interpretation $[\![f]\!]$ as the identity function over some set, but to state which, we must first define the function $\mathcal{M}$ that maps the term A to a set $\mathcal{M}_A$, and then define $[\![f]\!]$ as the identity function over the set $\mathcal{M}_A$.

In Predicate logic and in Deduction modulo theory, terms may be typed with sorts, but the sorts themselves have no type. In the $\lambda\Pi$-calculus modulo theory, in contrast, terms have types that have types... This explains that, in some cases, constructing the function $\mathcal{M}$ itself requires to define first another function $\mathcal{N}$, that is used as a tool helping to define this function. This can be iterated to a several layer model, where the function $[\![.]\!]$ is defined with the help of a function $\mathcal{M}$, that is defined with the help of a function $\mathcal{N}$, that is defined with the help... The number of layers depends on the model. Such layered constructions are common in proofs of termination of proof reduction, for instance for Pure Type Systems where sorts are stacked: $Type_0 : Type_1 : Type_2 : Type_3$.

Note that, in this definition of the notion of model, when a term t has type A, we do not require $[\![t]\!]_\phi$ to be an element of $[\![A]\!]_\phi$, but of $\mathcal{M}_A$. This is consistent with the notion of model of many-sorted predicate logic, where we require $[\![t]\!]_\phi$ to be an element of $\mathcal{M}_s$ and where $[\![s]\!]_\phi$ is often not even defined.

Super-consistency and proof reduction

The third goal of this talk is to use this notion of Π-algebra to define a notion of super-consistency and to prove that proof reduction, that is β-reduction, terminates in the $\lambda\Pi$-calculus modulo any super-consistent theory. We prove this way the termination of proof reduction in several theories expressed in the $\lambda\Pi$-calculus modulo theory, including Simple type theory and the Calculus of constructions. Together with confluence, this termination of proof reduction is a property required to define these theories in the system DEDUKTI.

Untyped Confluence In Dependent Type Theories

Ali Assaf[1], Gilles Dowek[2], Jean-Pierre Jouannaud[3], and Jiaxiang Liu[4]

[1] Google Inc.
[2] Inria and ENS Paris-Saclay
[3] École polytechnique and Université Paris-Saclay
[4] Tsinghua University

The two essential properties of a type theory, consistency and decidability of type checking, follow from three simpler ones: type preservation, strong normalization and confluence. In dependent type theories however, confluence and type preservation are needed to build strong normalization models; confluence is needed to show preservation of product types by rewriting, an essential ingredient of the type preservation proof; type preservation is needed to show that derivations issued from well-typed expressions are well-typed, an essential ingredient of the confluence proof. One can break this circularity in two ways: by proving all properties together within a single induction; or by proving confluence on untyped terms first, and then successively type preservation, confluence on typed terms, and strong normalization. The latter way is developed here. Its difficulty is that termination cannot be used anymore when proving confluence, hence forbidding the use of Newman's Lemma.

The confluence problem is indeed crucial for type theories allowing for user-defined computations such as Dedukti and now Agda. Current techniques for showing confluence by using van Oostrom theorem for higher-order rewrite systems are restricted to theories in which the rules are left linear, have development closed critical pairs, and do not build associativity and commutativity into pattern matching. But allowing for non-left-linear rules or for non-trivial critical pairs, and computing over non-free data structures whether first-order like sets or higher-order like abstract syntax, is out of scope of current techniques. Such computations are however present in Dedukti[1]. A main ambition of Dedukti is to serve as a common language for representing proof objects originating from different proof systems. Encoding these proof systems makes heavy use of the rewriting capabilities of $\lambda\Pi$Mod, the formal system on which Dedukti is based.

We describe a rewrite-based encoding in $\lambda\Pi$Mod of the *Calculus of Constructions with cumulative Universes* $\mathrm{CCU}^{\infty}_{\subseteq}$, which uses Nipkow's higher-order rewriting, non-left-linear rules, and associativity, commutativity and identity. $\mathrm{CCU}^{\infty}_{\subseteq}$ is a generalization of the calculus of constructions with an infinite hierarchy of predicative universes above the impredicative universe Prop. Together with inductive types, it forms the core of the *Calculus of Inductive Constructions* as is implemented in the proof system Coq. The major difficulty when encoding $\mathrm{CCU}^{\infty}_{\subseteq}$ is the treatment of *universe cumulativity*, which needs to be rendered explicit. Existing encodings of universe cumulativity in $\lambda\Pi$Mod have limitations which restrict their use to encode type systems, like Matita, which do not include universe polymorphism. Our rewrite based encoding is confluent on terms with variables, hence can support Coq's universe polymorphism.

Our major contribution is a result reducing the Church-Rosser property of a $\lambda\Pi$Mod theory on untyped terms to critical pair computations. This result goes far beyond the most advanced available technique based on van Oostrom's development closed critical pairs, which cannot handle our example. This result is applied to the previous encoding. It can be used more generally to show confluence of dependent type theories like those definable in Dedukti. Further, since the type system does not play any role in the confluence proof, the result applies as well to many calculi, typed or untyped, other than $\lambda\Pi$Mod.

[1]The system Dedukti is described at http://dedukti.gforge.inria.fr/

The main technical tool we use is van Oostrom decreasing diagrams for labelled relations, which permits to prove confluence of rewrite systems that verify a kind of local confluence property called decreasing diagram: local peaks need to be joinable by rewrites whose labels are smaller, in some well-founded sense, than those of the local peak. This method generalizes Hindley-Rosen's strong confluence lemma in the sense that a single rewrite step on each side of the joinability proof, the so-called *facing step*, can reproduce the label of the local peak's rewrite step it faces. This technique provides a modular analysis of local peaks by reflecting the various components of a rewrite system in the labels. In the case of $\lambda\Pi$Mod, we classify the rules and equations into three categories: the functional ones inherited from the λ-calculus; a set of user-defined first-order rules and equations forming a Church-Rosser, terminating, normal rewrite system; and a set of user-defined higher-order rules whose left-hand sides are patterns. Our definition of higher-order rewriting on untyped terms adapts Nipkow's definition given for typed terms by replacing β-normalization by Miller's β^0-normalization, which, unlike β, is terminating on untyped terms. Some β-steps that are implicit in Nipkow's must therefore become explicit in our setting.

Obtaining Church-Rosser calculi by putting together different confluent systems is known to be difficult in presence of non-left-linear rules. Further, confluence of arbitrary non-left-linear rules is never preserved in presence of a fixpoint combinator, which can itself be encoded in the pure λ-calculus. Variables having multiple occurrences in lefthand sides of user's rules are guaranteed to operate on homogeneous algebraic terms by a syntactic assumption, *confinement*: by ensuring that no redexes other than first-order ones may occur below a non-linear variable of a rule, confinement eliminates heterogeneous local peaks that would not have decreasing diagrams otherwise, like those occurring in Klop's fixpoint combinator example. This crucial new concept is built directly in the syntax of (pure) expressions.

This work shows an application of van Oostrom's decreasing diagrams method to a complex example in type theory, the very first of that kind. In our example, universes are specified as a confined type equipped with $0, 1$, addition, maximum and lifting. The associated first-order rewriting system is dubbed *normal* in the literature because expressions must be kept in normal form by eliminating the identity 0 in sums. Higher-order rules are used to describe the encoding of product types as well as of the decoding function. Below is a flavour of these various rules:

$$\begin{array}{rl}
1: & \max(i, i+j) \longrightarrow i+j \\
10: & \pi(i, i+j+1, a, \lambda x : T(i+j+1, a). \uparrow(i+j, (b\,x))) \\
 & \qquad\qquad \longrightarrow \uparrow(i+j, \pi(i, i+j, a, b)) \\
19: & \mathrm{T}(i+j+1, \pi(i+j+1, j+1, a, b)) \longrightarrow \\
 & \qquad\qquad \Pi x : T(i+j+1, a). T(j+1, (b\ x))
\end{array}$$

We can see here that the non-confined variable a in rule 10 is non-left-linear. This difficulty is solved by abstracting the type $T(i+j+1, a)$ of x by a new variable Y, a transformation that implies the confluence of the typed rules. Despite that many rules are higher-order, the critical pairs of the system have been computed by using MAUDE's confluence checker developed by José Meseguer's group at UIUC. Decreasing diagrams have then been obtained by hand.

Lower End of the Linial-Post Spectrum

Andrej Dudenhefner[1] and Jakob Rehof[1]

Technical University of Dortmund, Dortmund, Germany
{andrej.dudenhefner, jakob.rehof}@cs.tu-dortmund.de

In this work, we shed some light on the lower end of the Linial-Post spectrum (deciding axiomatizations of propositional calculi) from the point of view of functional program synthesis. For this purpose, we show that recognizing axiomatizations of the Hilbert-style calculus containing only the axiom scheme $\alpha \to \beta \to \alpha$ is undecidable. More importantly, the problem remains undecidable considering only *principal* axiom schemes, i.e. axiom schemes that, when seen as simple types, are principal for some λ-terms in normal form.

This result is motivated by two questions, which distinguish it from existing work (for an overview article see [7]). First, we want to explore the lower end of the Linial-Post spectrum. Existing work focuses on classical logic [4, 6] or superintuitionistic calculi [3, 7], often having rich type syntax, e.g. containing negation. In this work, we consider only implicational formulae and stay below $\alpha \to \beta \to \alpha$ in terms of derivability. This is arguably "as low as you can get" because recognizing axiomatizations of the Hilbert-style calculus with only the axiom scheme $\alpha \to \alpha$ is trivial. Second, we are interested in synthesis of functional programs from given building blocks. Following the same motivation as [1] and the line of work outlined in [5], we want to utilize proof search to synthesize code. Specifically, provided simply typed lambda terms $M_1, \ldots, M_n$ in normal form, we search for an applicative composition of the given terms that has some fixed simple type σ. This is equivalent to proof search in the Hilbert-style calculus having axiom schemes $\sigma_1, \ldots, \sigma_n$ where σ_i is the principal type of M_i for $i = 1 \ldots n$. It is a typical synthesis scenario, in which $M_1, \ldots, M_n$ are library components exposing functionality specified by their corresponding principal types. The synthesized composition of the given terms is a functional program that uses the library to realize behavior specified by the type σ.

Our second motivation forces us to deviate from standard constructions. In particular, for axiom schemes such as $\alpha \to \alpha \to \alpha$ (testing equality of two arguments), $(\alpha \to \beta) \to (\alpha \to \beta)$ (adding structure to $\alpha \to \alpha$) or $(\alpha \to \beta) \to \beta$ (encoding disjunction) there is no λ-term in $\beta\eta$-normal form having the corresponding axiom scheme as its principal type. Therefore, such logical formulae could be considered artificial and a pure logical peculiarity from the point of view of program synthesis. We identify "non-artificial" axiom schemes as *principal* by the following Definition 1.

Definition 1. *We say an axiom scheme σ is* principal *if there exists a λ-term M in $\beta\eta$-normal form such that σ is the principal type of M in the simply typed λ-calculus.*

Note that principality of a given axiom scheme is decidable [2]. In the following we always assume that any principal axiom scheme is given together with the corresponding λ-term.

Let us denote by $\{\sigma_1, \ldots, \sigma_n\} \vdash \sigma$ derivability of σ in the Hilbert-style calculus containing exactly the axiom schemes $\sigma_1, \ldots, \sigma_n$. Our main result is the following Theorem 2.

Theorem 2. *Given principal axiom schemes $\sigma_1, \ldots, \sigma_n$ such that $\{\alpha \to \beta \to \alpha\} \vdash \sigma_i$ for $i = 1 \ldots n$, it is undecidable whether $\{\sigma_1, \ldots, \sigma_n\} \vdash \alpha \to \beta \to \alpha$.*

Corollary 3. *Given λ-terms $M_1, \ldots, M_n$ in $\beta\eta$-normal form having principal types $\sigma_1, \ldots, \sigma_n$ in the simply typed λ-calculus such that $\{\alpha \to \beta \to \alpha\} \vdash \sigma_i$ for $i = 1 \ldots n$, it is undecidable whether there is an applicative composition of $M_1, \ldots, M_n$ having the simple type $\alpha \to \beta \to \alpha$.*

We prove Theorem 2 by reduction from (a slight variation of) the Post correspondence problem. Specifically, given pairs $(v_1, w_1), \dots, (v_k, w_k)$ of words over the alphabet $\{a, b\}$ such that $\epsilon \neq v_i \neq w_i \neq \epsilon$ for $i = 1 \dots k$, it is undecidable whether there exists an index sequence $i_1, \dots i_n$ such that $v_1 v_{i_1} v_{i_2} \dots v_{i_n} = w_1 w_{i_1} w_{i_2} \dots w_{i_n}$.

We encode a fixed Post correspondence problem instance as follows. For a word $v \in \{a, b\}^*$ we define its representation as $[v] = \alpha \cdot v$ where $\cdot$ is defined as

$$\sigma \cdot \epsilon = \sigma \qquad \sigma \cdot wa = (\alpha \to \alpha) \to (\sigma \cdot w) \qquad \sigma \cdot wb = (\alpha \to \alpha \to \alpha) \to (\sigma \cdot w)$$

We represent a pair of types σ, τ as

$$\langle \sigma, \tau \rangle = (\alpha \to \alpha) \to (\sigma \to \tau \to \alpha) \to (\alpha \to \sigma) \to (\alpha \to \tau) \to \alpha \to \alpha \to \alpha$$

Finally, we define a set Γ of $k + 2$ combinators typed by principal axiom schemes

$$\Gamma = \{F : \langle [v_1], [w_1] \rangle \to \alpha \to \beta \to \alpha, E : \langle \beta, \beta \rangle\} \cup \{C_i : \langle \beta \cdot v_i, \gamma \cdot w_i \rangle \to \langle \beta, \gamma \rangle \mid 1 \leq i \leq k\}$$

It appears to be difficult to pursue a constructive approach, i.e. start with (v_1, w_1), then append pairs of corresponding words and finally check for equality. Therefore, we pursue a "deconstructive" approach, i.e. start with an arbitrary pair of equal words (cf. E), then remove corresponding suffixes (cf. C_i) and finally compare with (v_1, w_1) (cf. F). Briefly, principality of the above axiom schemes is achieved by providing several key elements in pair representation $\langle \sigma, \tau \rangle$. First, $\alpha \to \sigma$ and $\alpha \to \tau$ provide constructors. Second, $\sigma \to \tau \to \alpha$ provides a verifier. Third, $\ldots \to \alpha \to \alpha \to \alpha$ provides branching. The component $\alpha \to \alpha$ is used as a "barrier". Specifically, a key property of the encoding is that there is no combinatory term e' such that $\Gamma \vdash e' : \sigma' \to \sigma'$ for some σ' and depth of e is smaller than the minimal depth of a combinatory term e with $\Gamma \vdash e : S(\langle [v_1], [w_1] \rangle)$ for some substitution S. As a result, we are able to enforce a linear shape on combinatory terms of small size which are typable in Γ and associate this shape with deconstructions of pairs of words. Lastly, the minimal inhabitant of $\alpha \to \beta \to \alpha$ in fact requires as a first argument a functional program that solves and verifies its solution to the given Post correspondence problem instance.

References

[1] Marcin Benke, Aleksy Schubert, and Daria Walukiewicz-Chrzaszcz. Synthesis of Functional Programs with Help of First-Order Intuitionistic Logic. In *FSCD 2016*, volume 52 of *LIPIcs*, pages 12:1–12:16, 2016.

[2] Sabine Broda and Luís Damas. Counting a Type's Principal Inhabitants. In *TLCA'99*, pages 69–82, 1999.

[3] AV Kuznecov and E Mendelson. Undecidability of the general problems of completeness, decidability and equivalence for propositional calculi. 1972.

[4] 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.

[5] Jakob Rehof. Towards Combinatory Logic Synthesis. In *BEAT'13, 1st International Workshop on Behavioural Types*. ACM, January 22 2013.

[6] Mary Katherine Yntema. A detailed argument for the Post-Linial theorems. *Notre Dame Journal of Formal Logic*, 5(1):37–50, 1964.

[7] Evgeny Zolin. Undecidability of the problem of recognizing axiomatizations of superintuitionistic propositional calculi. *Studia Logica*, 102(5):1021–1039, 2014.

Characterization of strong normalizability for a lambda-calculus with co-control

José Espírito Santo[1]*and Silvia Ghilezan[2,3]†

[1] Centro de Matemática, Universidade do Minho, Portugal
[2] University of Novi Sad, Serbia
[3] Mathematical Institute SANU, Serbia

We study strong normalization in the system $\overline{\lambda}\widetilde{\mu}$, a lambda calculus of proof-expressions with co-control for the intuitionistic sequent calculus [7]. In this sequent lambda calculus, the management of formulas on the left hand side of typing judgements is "dual" to the management of formulas on the right hand side of the typing judgements in Parigot's $\lambda\mu$-calculus [6] - that is why our system has first-class "co-control".

The abstract syntax of the untyped $\overline{\lambda}\widetilde{\mu}$ is given by the following grammar:

$$\begin{array}{rrcl} \text{(Terms)} & t, u, v & ::= & \lambda x.t \,|\, x\hat{\ }k \,|\, tk \\ \text{(Generalized vectors)} & k & ::= & [] \,|\, \tilde{\mu}x.v \,|\, u :: k \end{array}$$

One way of seeing this syntax is as a formal, generalized, relaxed, vector notation for λ-terms - while the original, informal, vector notation consists of the forms $\lambda x.M$, $x\vec{N}$, and $(\lambda x.M)N\vec{N}$. Another motivation for this syntax is through the type system for assigning simple types [7], which derives judgements $\Gamma \vdash t : A$ and $\Gamma|A \vdash k : B$. The constructions $[]$, $u :: k$, $\lambda x.t$ and tk correspond to the logical principles axiom, left and right introduction of implication, and cut. In addition, there is the $\tilde{\mu}$-operator [2], together with a construction $x\hat{\ }k$, already found in the $\overline{\lambda}$-calculus [4], to make antecedent formulas active or passive, respectively.

Among the reduction rules of $\overline{\lambda}\widetilde{\mu}$, the novelty lies in the rule for the $\tilde{\mu}$-operator, which makes use of a dual concept of structural substitution - that is, the structural substitution of a "co-continuation" $\mathcal{H}$ for a proof variable x. In the ordinary λ-calculus, and in $\lambda\mu$, we may think of the context $[\cdot]N_1 \cdots N_m$ as a continuation. In $\overline{\lambda}\widetilde{\mu}$, a co-continuation is a context with one of the forms $x\hat{\ }(u_1 :: \cdots :: u_m :: [\cdot])$ or $t(u_1 :: \cdots :: u_m :: [\cdot])$. The operator $u :: k$ is "right associative", so the hole $[\cdot]$ is again under a chain of arguments, but at the opposite end of the expression. The crucial equation in the recursive definition of structural substitution is $[\mathcal{H}/x](x\hat{\ }k) = \mathcal{H}[[\mathcal{H}/x]k]$. After this preparation, the reduction rule for $\tilde{\mu}$ reads $\mathcal{H}[\tilde{\mu}x.t] \to [\mathcal{H}/x]t$.

In $\overline{\lambda}\widetilde{\mu}$, the rule for $\tilde{\mu}$ coexists with four other reduction rules. Together, these rules reduce expressions of $\overline{\lambda}\widetilde{\mu}$ to a form corresponding to the cut-free proofs of LJT [4] - hence, the reduction rules express a combination of cut-elimination and focalization [5]. Both aspects are combined already in the reduction rule for $\tilde{\mu}$, because the rule eliminates all occurrences of the $\tilde{\mu}$-operator, hence its normal forms are the $\overline{\lambda}$-terms.

Intersection types are a powerful tool for characterizing strong normalization in different frameworks [1]. We employ them to obtain a characterization of strong normalizability in $\overline{\lambda}\widetilde{\mu}$ as typability in the system given in Fig. 1. In this figure, A, B, C range over intersection types and $\cap A_i = A_1 \cap \cdots \cap A_n$. We work with types modulo an equivalence relation, generated by a standard preorder $\leq$. In particular, our intersection types are associative, commutative and idempotent.

*This work was supported by FCT—Fundação para a Ciência e a Tecnologia, within the project UID-MAT-00013/2013, and by COST Action CA15123 - The European research network on types for programming and verification (EUTypes) via STSM.

†This work was partly supported by the grants ON174026 and III44006 of the Ministry of Education and Science, Serbia.

Figure 1: The intersection type assignment system of $\overline{\lambda\tilde{\mu}}$

$$\frac{\exists i \in \{1,\cdots,n\}\; A = A_i}{\Gamma | \cap A_i \vdash [\,] : A}\;(Ax)$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\;(\to_R) \qquad \frac{\Gamma \vdash t : A_i,\;\; \forall i \in \{1,\cdots,n\} \qquad \Gamma|B \vdash k : C}{\Gamma | \cap A_i \to B \vdash t :: k : C}\;(\to_L)$$

$$\frac{\Gamma \vdash t : A_i,\; \forall i \in \{1,\cdots,n\} \qquad \Gamma | \cap A_i \vdash k : B}{\Gamma \vdash tk : B}\;(Cut)$$

$$\frac{\Gamma, x : \cap A_i | \cap A_j \vdash k : B \quad \forall j\, \exists i \in \{1,\cdots,n\},\, A_j = A_i}{\Gamma, x : \cap A_i \vdash x\hat{}k : B}\;(Pass) \qquad \frac{\Gamma, x : A \vdash v : B}{\Gamma | A \vdash \tilde{\mu}x.v : B}\;(Act)$$

The system we propose for $\overline{\lambda\tilde{\mu}}$ is obtained by adapting the system for assigning intersection types used to characterize the strongly normalizing proof-terms of λGtz, in previous work by the authors and colleagues [8, 9]. The λGtz-calculus [3] is another sequent lambda calculus, where the treatment of the $\tilde{\mu}$-operator follows the original and simpler one found in [2]: the $\tilde{\mu}$-operator is a term-substitution former, and its reduction principle triggers an ordinary term substitution. The characterization of strong normalizability in $\overline{\lambda\tilde{\mu}}$ is proved, not by re-running the proof for λGtz, but by "lifting" the characterization in λGtz. This requires a detailed comparison of the two rewriting systems, which is of independent interest, as it highlights sensitive choice points in the design of calculi of proof terms for the sequent calculus, particulary the treatment of proof-term variables and the related substitution principles.

References

[1] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[2] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM, 2000.

[3] José Espírito Santo. The λ-calculus and the unity of structural proof theory. *Theory of Computing Systems*, 45:963–994, 2009.

[4] H. Herbelin. A λ-calculus structure isomorphic to a Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *Proceedings of CSL'94*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 1995.

[5] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logic. *Theoretical Computer Science*, 410:4747–4768, 2009.

[6] Michel Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, pages 190–201, 1992.

[7] José Espírito Santo. Curry-Howard for sequent calculus at last! In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 165–179. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[8] José Espírito Santo, Silvia Ghilezan, and Jelena Ivetic. Characterising strongly normalising intuitionistic sequent terms. In Marino Miculan, Ivan Scagnetto, and Furio Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, volume 4941 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2008.

[9] José Espírito Santo, Jelena Ivetic, and Silvia Likavec. Characterising strongly normalising intuitionistic terms. *Fundam. Inform.*, 121(1-4):83–120, 2012.

Inhabitation in Simply-Typed Lambda-Calculus through a Lambda-Calculus for Proof Search

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), C.N.R.S. and Univ. of Toulouse, France

This work (of which details can be found on the arXiv [4]) is about an analysis of the inhabitation problem in simply-typed λ-calculus by way of a λ-calculus notation for proof search in minimal implicational logic, introduced by the authors [2] (see also the revised and extended version [3]). By proofs in minimal implicational logic we understand η-long β-normal λ-terms that are well-typed according to the rules of simply-typed λ-calculus. One has to treat the general case of terms with open assumptions: a sequent σ is of the form $\Gamma \Rightarrow A$ with a finite set Γ of declarations $x_i : A_i$, where the x_i are variables of λ-calculus. This fits with the typing relation of λ-calculus but, viewed from the logical side, presents the particularity of named hypotheses. The total discharge convention that plays a role in the paper by Takahashi et al [6] goes into the opposite direction and considers λ-terms where there is only one term variable per type. In the previous work of the present authors, cited above, no total discharge convention is needed for obtaining a finitary description of the whole solution space $\mathcal{S}(\sigma)$ for a given sequent σ. The solution space $\mathcal{S}(\sigma)$ itself is a *coinductive* expression formed from the grammar of β-normal forms of λ-calculus and an operator for finite sums expressing proof alternatives. Its potential infinity reflects the *a priori* unlimited depth of proof search and serves the specification of proof search problems. For simply-typed λ-calculus, the subformula property allows to describe the solution spaces finitely. This may be seen as a coinductive extension of work done already by Ben-Yelles [1] with a very concrete λ-calculus approach and by Takahashi et al [6] by using formal grammar theory (but the latter need the total discharge convention for reaching finiteness). The announced λ-calculus notation for proof search is thus [2, 3]:

$$\begin{array}{llcl} \text{(terms)} & N & ::= & \lambda x^A.N \mid \mathsf{gfp}\ X^\sigma.E_1 + \cdots + E_n \mid X^\sigma \\ \text{(elimination alternatives)} & E & ::= & x\langle N_1 \ldots N_k\rangle \end{array}$$

where X is assumed to range over a countably infinite set of *fixpoint variables*, and the sequents σ are supposed to be atomic, i. e., with atomic conclusion. A fixpoint variable may occur with different sequents in a term, but only *well-bound* terms are generated when building a finitary representation of $\mathcal{S}(\sigma)$, and only well-bound terms T are given a semantics $[\![T]\!]$ as a coinductive term. In essence, a term is well-bound if the fixed-point operator gfp with X^σ only binds free occurrences of $X^{\sigma'}$ where the σ' are inessential extensions of σ in the sense that they have the same conclusion and maybe more bindings, but only with types/formulas that already have a binding in σ. The main result is that there is a term $\mathcal{F}(\sigma)$ without free fixpoint variables (called *closed* term) whose semantics is $\mathcal{S}(\sigma)$ (modulo a notion of bisimilarity that considers the sums of alternatives as sets) [2].

In this work we show the applicability of our term representation $\mathcal{F}(\sigma)$ to the analysis of inhabitation problems. Consider a decision problem D and let A be a type. Our previous work

*The first and the third author were partially supported by FCT—'Fundação para a Ciência e a Tecnologia', within the project UID-MAT-00013/2013.

†This work was financed by the project *Climt*, ANR-11-BS02-016, of the French Agence Nationale de la Recherche.

[2, 3] allowed us (i) to express $D(A)$ as $P(S_A)$, where S_A is the coinductive description of the search for inhabitants of A (specifically $\mathcal{S}(\Rightarrow A)$), and P is some coinductive predicate, and then (ii) to convert to the equivalent $P'(F_A)$, where F_A is the finitary description of S_A (specifically $\mathcal{F}(\Rightarrow A)$) and P' is still a predicate defined by reference to coinductive structures. The form $P'(F_A)$ does not yet profit from the finitary description. The first task of the present work is, simultaneously, its main technical achievement: one obtains the equivalent $P''(F_A)$, where P'' is inductive, actually directed by the syntax of the finitary description. The decidability of P'' (whence of D) is an immediate bonus.

One of the problems to which we applied our methodology is the classical decision problem: "does type A have an inhabitant?". In order to do so, we defined complementary predicates exfin (inductively) and nofin (coinductively) capturing the existence (resp. non-existence) of finite solutions (i. e., proofs/inhabitants) in a solution space $\mathcal{S}(\sigma)$. Then, we defined inductively the predicate EF_P on the finitary expressions

$$\frac{P(\sigma)}{\mathsf{EF}_P(X^\sigma)} \qquad \frac{\mathsf{EF}_P(N)}{\mathsf{EF}_P(\lambda x^A.N)} \qquad \frac{\mathsf{EF}_P(E_j)}{\mathsf{EF}_P(\mathsf{gfp}\, X^\sigma . \sum_i E_i)} \qquad \frac{\forall i,\ \mathsf{EF}_P(N_i)}{\mathsf{EF}_P(x\langle N_i\rangle_i)}$$

s. t., for appropriately restricted finitary terms T, a (partly) coinductive reasoning shows $\mathsf{EF}_P(T)$ iff $\mathsf{exfin}([\![T]\!])$, where P is a predicate on sequents s. t. $P(\sigma)$ implies $\mathsf{exfin}(\mathcal{S}(\sigma))$. Instantiation of P to the empty predicate allows then to decide the predicate $\mathsf{exfin}(\mathcal{S}(\sigma))$ (given σ), and hence the classical inhabitation problem, through a simple procedure recursing over the structure of the finitary term $\mathcal{F}(\sigma)$.

We also applied our methodology to the decision problem: "does type A have finitely many inhabitants?" (considered in [6]), and this works smoothly, along the lines described above for the classical inhabitation problem. Crucially, in subsequent definitions of auxiliary predicates, we reuse the above predicate, with P instantiated to $\mathsf{exfin} \circ \mathcal{S}$. Additionally, we used our tools to study types A with finitely many inhabitants: Ben-Yelles' algorithm [1, 5] can count them, here we show how their number can be calculated from the finitary description $\mathcal{F}(\Rightarrow A)$ as a rather simple recursive function.

We remark that a previous version of our results took the wrong decision for the instantiation of predicate P in EF_P, and there was an error in the proof (that did not work either for the correct instantiation), and so we had to rework the proof method by introducing a simplified semantics for the purpose of proof. As mentioned before, the details are available at arXiv [4].

References

[1] Choukri-Bey Ben-Yelles. Type assignment in the lambda-calculus: syntax & semantics. University of College of Swansea, 1979.

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

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

[4] José Espírito Santo, Ralph Matthes, and Luís Pinto. Inhabitation in simply-typed lambda-calculus through a lambda-calculus for proof search. `http://arxiv.org/abs/1604.02086v2`, March 2017.

[5] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[6] Masako Takahashi, Yohji Akama, and Sachio Hirokawa. Normal proofs and their grammar. *Inf. Comput.*, 125(2):144–153, 1996.

Dependent Type Theory with Contextual Types

Francisco Ferreira, David Thibodeau, and Brigitte Pientka

McGill University, Montréal, Québec, Canada
`fferre8@cs.mcgill.ca, david.thibodeau@mail.mcgill.ca, bpientka@cs.mcgill.ca`

We present a type theory with support for Higher Order Abstract Syntax (HOAS) by extending Martin-Löf style type theory [4] with contextual types and a specification framework based on the logical framework LF [3]. This system can be seen as the extension of the theory of Beluga [6] into full dependent types. The resulting system supports the definition of abstract syntax with binders and the use of substitutions. This simplifies proofs about systems with binders by providing for free the substitution lemma and lemmas about the equational theory of substitutions. As in Beluga, we mediate between specifications and computations via contextual types. However, unlike Beluga, we can embed and use computations directly in contextual objects and types, hence we allow the arbitrary mixing of specifications and computations following ideas from [2]. Moreover, dependent types allow for reasoning about proofs in addition to reasoning about LF specifications.

The syntax of calculus is presented in a Pure Type Systems [1] with only one grammar as computations and specifications are interleaved arbitrarily. For clarity, we use the following naming convention for terms. We use E for expressions, S and T for types, Ψ and Φ for specification-level contexts, α and β for specification types, and σ and ϕ for substitutions.

Terms		
$E, S, T, \Psi, \Phi, \sigma, \alpha, \beta$	$::= \mathtt{set}_n \mid (x : S) \to T \mid \lambda x.E \mid E\, E' \mid x \mid \mathbf{c} \mid [\Psi \vdash E]$	T.T. terms
	$\mid\ [\Psi \vdash \alpha] \mid \mathtt{ctx}$	Cont. types
	$\mid\ \star \mid (\widehat{x} : \alpha) \twoheadrightarrow \beta$	Spec. types
	$\mid\ \widehat{\lambda} x.E \mid E \,'\, E' \mid E[\sigma] \mid \widehat{x} \mid ^\wedge \mid \mathtt{id} \mid \sigma; \widehat{x} := E \mid \emptyset \mid \Psi, \widehat{x} : E$	Spec. terms
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : T$	
Signature	$\Sigma ::= \cdot \mid \Sigma, \mathbf{c} : T$	

We present a Martin-Löf style type theory with an infinite hierarchy of universes extended with contextual types $[\Psi \vdash \alpha]$ which represent a specification type α in an open context Ψ. Specification types are classified by a single universe $\star$, together with an intensional function space $(\widehat{x} : \alpha) \twoheadrightarrow \beta$ and constants. Substitutions can be an empty substitution $^\wedge$ which weakens closed terms, an identity substitution `id`, or a substitution extended with a term for a variable $\sigma; \widehat{x} := E$. Contexts are either empty $\emptyset$ or a context extended with a new assumption $\Psi, \widehat{x} : E$.

Type theory terms and specification terms are typed with two different judgments $\Gamma \vdash E : T$ and $\Gamma; \Psi \vdash E : \beta$, respectively. For instance, we type the type theory functions and specifications functions in the following way:

$$\frac{\Gamma, x : S \vdash E : T}{\Gamma \vdash \lambda x.E : (x : S) \to T}\ \texttt{t-fun} \qquad \frac{\Gamma; \Psi, \widehat{x} : \alpha \vdash E : \beta}{\Gamma; \Psi \vdash \widehat{\lambda} x.E : (\widehat{x} : \alpha) \twoheadrightarrow \beta}\ \texttt{s-lam}$$

The two function spaces differ by the contexts they act on. Type theory λ-abstractions introduce variables in the computational context Γ while specification λ-abstractions use the specification context Ψ. The β-rule for each λ-abstraction uses its own substitution operation for its corresponding class of variables denoted respectively $\{\theta\}E$ and $[\sigma]E$. This is in contrast to $E[\sigma]$ for the closure that embeds a computation of boxed type into a specification and is defined by the following introduction rule:

$$\frac{\Gamma ; \Psi \vdash \sigma : \Phi \quad \Gamma \vdash E : [\Phi \vdash \alpha]}{\Gamma ; \Psi \vdash E[\sigma] : [\sigma]\alpha} \texttt{s-clo}$$

In addition to the description of the term calculus we present a prototype implementation[1] for our type theory which supplements the calculus with recursive functions and Agda-style dependent pattern matching [5] extended to allow matching on specifications including on substitutions and specification-level λ-abstractions and thus abstracting over binders.

For example, we can write a type preserving translation on intrinsically typed terms where we express the notion of type preservation using a computational function `tran-tp` translating the types.

```
def tran-tp : (⊢ s-tp) → (⊢ t-tp) where ...

data rel: ctx → ctx → set where
| empty : rel ∅ ∅
| cons : (g : ctx) → (h : ctx) → (t : ⊢ s-tp) → rel g h
       → rel (g, x: s-tm ’ t) (h, y: t-tm ’ (tran-tp t))

def tran : (g : ctx) → (h : ctx) → rel g h → (t : ⊢ s-tp)
         → (g ⊢ s-tm ’ t) → (h ⊢ t-tm ’ (tran-tp t)) where ...
```

We use the `rel` predicate to relate the source and target contexts. The translation then moves from a term in the source language of type `t` to a term in the target language of translated type `tran-tp t`.

In conclusion, we developed a theory that allows for embedding contextual LF specifications into a fully dependently typed language that simplifies proofs about structures with syntactic binders (such as programming languages and logics). Moreover, we have a prototype, the Orca system, in which we implement some example proofs.

References

[1] Stefano Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Dept. of Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, 1988.

[2] Francisco Ferreira and Brigitte Pientka. Programs using syntax with first-class binders. In Hongseok Yang, editor, *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Uppsala, Sweden*, pages 504–529. Springer Berlin Heidelberg, 2017.

[3] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[4] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory Lecture Notes*. Bibliopolis, Napoli, 1984.

[5] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, September 2007. Technical Report 33D.

[6] Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming proofs. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 272–281, 2015.

[1]Available at:http://github.com/orca-lang/orca

Inductive-Recursive Definitions and Composition

Neil Ghani[1], Conor McBride[1], Fredrik Nordvall Forsberg[1], and Stephan Spahn[2]

[1] University of Strathclyde, Glasgow, Scotland
{neil.ghani, conor.mcbride, fredrik.nordvall-forsberg}@strath.ac.uk
[2] Middlesex University, London, England
stephanspahn1@me.com

Inductive-recursive definitions. The idea behind inductive-recursive definitions is to make an inductive definition of a data type U at the same time as defining a function $T : U \to D$ which computes by recursion over that type; here, "at the same time" means that the function can occur in the type of the constructors for the inductive type. Typically, induction-recursion is used to construct an encoding (defined inductively) where some data depends on the *meaning* of previous data (computed recursively), with the prototypical example being a universe à la Tarski [Pal98].

Dybjer and Setzer wrote a series of papers [DS99, DS03, DS06] characterising inductive-recursive definitions in Martin-Löf Type Theory. A pair (U, T) as above where $U : \mathsf{Set}$ and $T : U \to D$ can be seen as an object in the category $\mathsf{Fam}(D) = \big(\Sigma I : \mathsf{Set}\big)\big(I \to D\big)$. Inductive-recursive definitions arise as initial algebras of certain functors $[\![c]\!] : \mathsf{Fam}(D) \to \mathsf{Fam}(D)$ that are given by codes c in a coding schema $\mathsf{IR}(D)$ mimicking the constructors of an inductive-recursively defined type. Let us call a functor $F : \mathsf{Fam}(D) \to \mathsf{Fam}(D)$ *IR-definable* if there is an IR code $c : \mathsf{IR}(D)$ such that $F(X) \cong [\![c]\!](X)$ for every $X : \mathsf{Fam}(D)$.

Closure under composition. Given two IR-definable functors, is their composite also IR-definable? A positive answer to this question would allow a certain modularity in data type definitions, in that one can modify inductive occurrences in the constructors by composing with another functor. From the point of view of inductive definitions as least solutions to type equations $F(X) \cong X$, this means that we would be able to solve more general equations of the form $F(G(X)) \cong X$ for IR-definable functors F and G. Besides, being closed under composition is a natural property to require of a class of functors.

Unfortunately, we do not know whether IR-definable functors are closed under composition or not. Altenkirch considered the problem already in 2011 [Alt11]. The main ingredients needed seem to be a way to graft one IR code at the end of another, and to be able to form infinite powers of IR codes representing $A \to [\![c]\!](X)$. The collection $\mathsf{IR}(D)$ of IR codes is monadic [GH16], in fact a free monad, and the bind of this monad gives a code grafting operation. However, it seems hard to define powers of codes.

A subsystem of uniform IR codes. To shed light on this issue, we consider a variation of inductive-recursive definitions originally due to Hancock. Here codes are given in a left-nested, not right-nested fashion [Pol02], and as a result have a uniform appearance. We call this system of codes *Uniform IR codes* and the functors they decode to *Uniform IR functors*. Uniform codes can be translated to ordinary Dybjer-Setzer IR codes in a meaning-preserving way, but it is not clear how to turn a non-uniform code into a uniform one in general. However, all inductive-recursive definitions that we know of "in the wild" are already given by uniform codes. The left-nestedness of uniform IR codes can be exploited to define a power operation (most economically expressed combined with grafting) by induction over the codes, and using this, we can define the composite of two codes:

Theorem 1. *Uniform IR functors are closed under composition, i.e. for all uniform IR codes c and d, there is a uniform IR code $c \bullet d$ such that $[\![c \bullet d]\!](X) \cong [\![c]\!]([\![d]\!](X))$ for every $X : \mathsf{Fam}(D)$.*

A supersystem of polynomial codes. Another option is to explicitly add a power operation to the system of codes. Doing this naively results in a system of codes which is no longer monadic, but the second author discovered a way to add powers (in fact dependent products) and still retain monadicity. Because the resulting system is given by sums and (set-indexed) dependent products, we call the codes *polynomial codes* and the corresponding functors *polynomial IR functors* (warning: not to be confused with polynomial functors [GK13]!). This time, Dybjer-Setzer codes embed into polynomial codes, but it does not seem possible to go the other way. On the other hand, again we know of no non-contrived examples of polynomial IR functors that we expect not to be Dybjer-Setzer IR-definable. Using powers and grafting, we have:

Theorem 2. *Polynomial IR functors are closed under composition, i.e. for all polynomial codes c and d, there is a polynomial code $c \bullet d$ such that $[\![c \bullet d]\!](X) \cong [\![c]\!]([\![d]\!](X))$ for every $X : \mathsf{Fam}(D)$.*

Open questions. Are the inclusions

$$\text{Uniform IR} \hookrightarrow \text{Dybjer-Setzer IR} \hookrightarrow \text{Polynomial IR}$$

strict? Are Dybjer-Setzer IR-definable functors closed under composition? Are they closed under powers? Most of these questions are in fact the same question.

Formalisation. A work-in-progress Agda formalisation of the various systems can be found at `http://personal.cis.strath.ac.uk/fredrik.nordvall-forsberg/variantsIR/`.

Acknowledgements We thank Peter Hancock and Anton Setzer for discussions and comments. This research was supported by EPSRC grants EP/K023837/1 and EP/M016951/1.

References

[Alt11] Thorsten Altenkirch. What is the problem with induction-recursion? Talk at Peter Hancock 60th Celebration, 2011. `http://www.cs.nott.ac.uk/~psztxa/talks/HankFest11.pdf`.

[DS99] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *TLCA*, pages 129–146. Springer Verlag, 1999.

[DS03] Peter Dybjer and Anton Setzer. Induction–recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1-3):1–47, 2003.

[DS06] Peter Dybjer and Anton Setzer. Indexed induction–recursion. *Journal of logic and algebraic programming*, 66(1):1–49, 2006.

[GH16] Neil Ghani and Peter Hancock. Containers, monads and induction recursion. *Mathematical Structures in Computer Science*, 26(1):89–113, 2016.

[GK13] Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154:153–192, 2013.

[Pal98] Erik Palmgren. On universes in type theory. In Giovanni Sambin and Jan Smith, editors, *Twenty five years of constructive type theory*, pages 191 – 204. Oxford University Press, 1998.

[Pol02] Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13(3):386–402, 2002.

Verifiable certificates for predicate subtyping

Frédéric Gilbert

École des Ponts ParisTech, Inria, CEA LIST
frederic.a.gilbert@inria.fr

The simple types of higher-order logic are sufficient to exclude from reasoning unexpected expressions such as the application of a predicate to itself. Extending higher-order logic with predicate subtyping allows to restrict the set of valid expression further, excluding for instance any expression containing a division by zero; it is used at the core of the type system of the proof assistant PVS [3]. The present work is focused on two type systems. The first one, referred to as PVS-Core, is a formalization of the extension of higher-order logic with predicate subtyping. Type-checking undecidable in this system. The second, referred to as PVS-Cert, is a type system with explicit proof terms, in which type-checking is decidable. The two systems are tightly connected, and we show how to use the terms of PVS-Cert as verifiable certificates for PVS-Core. We also show that both systems are conservative extensions of higher-order logic.

The extension of higher-order logic with predicate subtyping is based on the addition of a new construction of types on top of the simple types of higher-order logic. This construction is denoted $\{x : A \mid P\}$, where A is expected to be a type and P is expected to be a predicate on A. For instance, in a context where a type for natural numbers *Nat* is defined, we can define the type $\mathit{Nonzero} = \{x : \mathit{Nat} \mid x \neq 0\}$ with predicate subtyping. In PVS-Core, there are no constructors nor eliminators for predicate subtypes: a term t admits the type $\{x : A \mid P\}$ if it admits the type A and $P[t/x]$ is provable. In particular, $\{x : A \mid P\}$ is a subtype of A.

Starting from a usual presentation of higher-order logic with a type of proposition *Prop* and three kinds of judgments, $\Gamma \vdash A : \mathit{Type}$ (well-formed types), $\Gamma \vdash t : A$ (well-typed terms), and $\Gamma \vdash P$ (provable proposition), we define PVS-Core by adding the following rules:

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash P : \mathit{Prop}}{\Gamma \vdash \{x : A \mid P\} : \mathit{Type}}\ \textsc{Subtype} \qquad \frac{\Gamma \vdash t : \{x : A \mid P\}}{\Gamma \vdash t : A}\ \textsc{Elim1}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash P[t/x] \qquad \Gamma \vdash \{x : A \mid P\} : \mathit{Type}}{\Gamma \vdash t : \{x : A \mid P\}}\ \textsc{Intro} \qquad \frac{\Gamma \vdash t : \{x : A \mid P\}}{\Gamma \vdash P[t/x]}\ \textsc{Elim2}$$

Moreover, the arrows are replaced by dependent products $\Pi x : A.B$. In higher-order logic, the only undecidable kind of judgment is $\Gamma \vdash P$, while, in PVS-Core, all kinds of judgments become undecidable because of the rules Intro and Subtype.

The second system, PVS-Cert, is an extension the PTS λHOL. λHOL is itself a language of certificates for higher-order logic, in the sense that provable propositions are inhabited by proof terms of λHOL, which can be used as certificates as type-checking is decidable in this system. λHOL admits three sorts *Prop*, *Type*, and *Kind*, two axioms *Prop* : *Type* and *Type* : *Kind*, and three rules $\langle \mathit{Prop}, \mathit{Prop}, \mathit{Prop} \rangle$, $\langle \mathit{Type}, \mathit{Type}, \mathit{Type} \rangle$, and $\langle \mathit{Type}, \mathit{Prop}, \mathit{Prop} \rangle$. We extend λHOL to PVS-Cert with the following rules:

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash P : \mathit{Prop}}{\Gamma \vdash \{x : A \mid P\} : \mathit{Type}}\ \textsc{Subtype} \qquad \frac{\Gamma \vdash t : \{x : A \mid P\}}{\Gamma \vdash \pi_1(t) : A}\ \textsc{Elim1}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash p : P[t/x] \qquad \Gamma \vdash \{x : A \mid P\} : \mathit{Type}}{\Gamma \vdash \langle t, p\rangle_{\{x:A|P\}} : \{x : A \mid P\}}\ \textsc{Intro} \qquad \frac{\Gamma \vdash t : \{x : A \mid P\}}{\Gamma \vdash \pi_2(t) : P[\pi_1(t)/x]}\ \textsc{Elim2}$$

We define the two rewrite rules $\langle M_1, M_2\rangle_A \rightarrow_* M_1$ and $\pi_1(M) \rightarrow_* M$, and replace the original conversion $\equiv_\beta$, with the conversion $\equiv_{\beta*}$ based on β-reduction extended with these rules. Using the result of [2], $\rightarrow_{\beta*}$ is known to be confluent.

The first important property of PVS-Cert is the stratification of terms: well-typed terms can be classified in four classes of terms: $\{\mathit{Type}\}$ and three other classes, which will be referred to as types, elements, and proofs. Through stratification, PVS-Cert and PVS-Core are related in a very simple way. Starting from a derivation in PVS-Cert, we obtain a derivation in PVS-Core by normalizing every term under $\rightarrow_*$ and erasing proof terms from the judgments. Conversely, the derivations from PVS-Core can be translated into PVS-Cert. Hence, PVS-Cert can be used as a system of proof certificates for PVS-Core as long as type-checking is decidable in it.

In order to prove the decidability of type-checking in PVS-Cert, we prove the strong normalization of $\rightarrow_{\beta*}$ and the existence of a type-preserving reduction allowing to transform any well-typed term convertible with a dependent product into a dependent product. This reduction cannot be defined with $\rightarrow_{\beta*}$, which is not type preserving. Instead, it is defined with $\rightarrow_{\beta\sigma}$, where $\pi_i\langle M_1, M_2\rangle_A \rightarrow_\sigma M_i$ for all $i \in \{1, 2\}$.

In this setting, the decidability of type-checking in PVS-Cert follows from the preservation of types of $\rightarrow_{\beta\sigma}$, the strong normalization $\rightarrow_{\beta*}$, and the strong normalization of $\rightarrow_{\beta\sigma}$. The preservation of types under $\rightarrow_{\beta\sigma}$ is established using the stratification property and the following lemma: if $M_1 \equiv_{\beta\sigma} M_2$ and M_i is either a type or an element for all $i \in \{1, 2\}$, then $M_1 \equiv_{\beta*} M_2$.

The two expected properties of strong normalization of $\rightarrow_{\beta\sigma}$ and $\rightarrow_{\beta*}$ cannot be merged into the strong normalization of $\rightarrow_{\beta\sigma*}$. Indeed, we present some well-typed term on which $\rightarrow_{\beta\sigma*}$ doesn't terminate. However, we prove both properties using a single definition of saturated sets. This notion of saturated sets, inspired from [1], is adapted to apply both to $\rightarrow_{\beta\sigma}$ and $\rightarrow_{\beta*}$. In order to handle both reductions more easily, these definitions are based on untyped terms instead of using encodings to pure lambda calculus.

Using the type preservation and the strong normalization of $\rightarrow_{\beta\sigma}$, we also prove that PVS-Cert is a conservative extension of higher-order logic. As a consequence, PVS-Core is a conservative extension higher-order logic as well. A possible future work is to implement PVS-Cert and to instrument the proof assistant PVS to export proof certificates in PVS-Cert.

References

[1] Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In International Workshop on Types for Proofs and Programs, pages 14–38. Springer, 1994.

[2] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. Theoretical Computer Science, 121(1):279 – 308, 1993.

[3] Sam Owre, John M Rushby, and Natarajan Shankar. PVS: A prototype verification system. In International Conference on Automated Deduction, pages 748–752. Springer, 1992.

[4] Matthieu Sozeau. Subset coercions in coq. In International Workshop on Types for Proofs and Programs, pages 237–252. Springer, 2006.

[5] Benjamin Werner. On the strength of proof-irrelevant type theories. In International Joint Conference on Automated Reasoning, pages 604–618. Springer, 2006.

Strong Bisimilarity Implies Trace Semantics in CSP-Agda

Bashar Igried and Anton Setzer

Swansea University, Swansea,Wales, UK
`bashar.igried@yahoo.com , a.g.setzer@swansea.ac.uk`

Abstract

In this talk we extend the library CSP-Agda with a semantics in order to prove properties of safety critical systems. In CSP-Agda the process algebra CSP is represented coinductively in the interactive theorem prover Agda. We present trace semantics in the theorem prover Agda, together with the corresponding refinement and equality relation. In order to facilitate proofs of algebraic properties for this semantics we introduce strong bisimilarity, and show that it implies trace equivalence. As an example we apply this methodology to commutativity of interleaving.

1 Introduction

Process algebras are amongst the most important and successful frameworks for modelling and specifying interactive systems. They have demonstrated to be convenient at the level of requirement and design specification, as well as in proofs of refinement. There are many process algebras available. One prominent process algebra is CSP which has been developed and widely used in many areas. In [1, 2] we introduced a library called CSP-Agda for representing processes in the dependently typed theorem prover and interactive programming language Agda. A process in CSP-Agda can either terminate, returning a result. Or it can be a tree branching over external and internal choices, where for each such choice a continuation process is given. Instead of forming processes by using high level operators, as it is usually done in process algebras, our processes are given by these atomic one step operations. The high level operators are defined operations on these processes. Since processes can loop for ever, processes are defined coinductively from this one step operation. The transitions, a process can make, are labelled external choices, silent τ-transitions, and termination events $\checkmark$. We add a return value to termination events, and therefore extend CSP processes to monadic ones. There are 3 levels of processes in CSP-Agda, Process∞, Process, Process+, which are defined mutually coinductively. Therefore predicates and functions on processes need to be defined for each of these 3 levels. For sake of brevity, we give only definitions referring to Process+.

The purpose of this paper is to extend CSP-Agda by adding (finite) trace semantics of CSP. and show how to prove selected (adjusted) algebraic laws of CSP in CSP-Agda.

2 Defining Trace Semantics for CSP-Agda

In CSP traces of a process are the sequences of actions or labels of external choices, a process can perform. Since the process in CSP are non-deterministic, a process can follow different traces during its execution. The corresponding definition for trace is as follows:

```
data Tr+ {c : Choice} : (l : List Label) → Maybe (ChoiceSet c) → (P : Process+ ∞ c)
   → Set where
empty : {P : Process+ ∞ c} → Tr+ [] nothing P
extc  : {P : Process+ ∞ c} → (l : List Label) → (tick : Maybe (ChoiceSet c))
```

```
      → (x : ChoiceSet (E P)) → Tr∞ l tick (PE P x) → Tr+ (Lab P x :: l) tick P
  intc  : {P : Process+ ∞ c} → (l : List Label) → (tick : Maybe (ChoiceSet c))
      → (x : ChoiceSet (I P)) → Tr∞ l tick (PI P x) → Tr+ l tick P
  terc  : {P : Process+ ∞ c} → (x : ChoiceSet (T P)) → Tr+ [] (just (PT P x)) P
```

For instance the constructor extc is used for defining the trace corresponding to external choice: if a process P has external choice x, then from every trace for the result of following this choice, which consisting of a list of labels l and a possible result $tick$, we obtain a trace of P consisting of the result of adding in front of l the label of that external choice, and of the same possible result $tick$. The resulting proof will be denoted by (extc l $tick$ x tr).

A process P refines a process Q, written ($P \sqsubseteq Q$) if and only if any observable behaviour of Q is an observable behaviour of P, i.e. if $traces(Q) \subseteq traces(P)$. They are trace equivalent, denoted by $P \equiv Q$, if they refine each other.

In CSP two process are strong bisimilar if they behave in the same way in the sense that one process simulates the other and vice versa. We define strong bisimilarity in CSP-Agda as follows:

```
  record Bisims+ {i : Size}{c : Choice} (P P' : Process+ ∞ c) : Set where
    coinductive
    field
      bisim2E    : (e : ChoiceSet (E P)) → ChoiceSet (E P')
      bisimELab  : (e : ChoiceSet (E P)) → Lab P e ≡ Lab P' (bisim2E e)
      bisimENext : (e : ChoiceSet (E P)) → Bisims∞ (PE P e) (PE P' (bisim2E e))
      bisim2I    : (i : ChoiceSet (I P)) → ChoiceSet (I P')
      bisimINext : (i : ChoiceSet (I P)) → Bisims∞ (PI P i) (PI P' (bisim2I i))
      bisim2T    : (t : ChoiceSet (T P)) → ChoiceSet (T P')
      bisim2TEq  : (t : ChoiceSet (T P)) → PT P t ≡ PT P' (bisim2T t)
```

For sake of brevity we gave in the above definition only the fields expressing that P' simulates P. In the full definition one needs to add seven more fields expressing that P simulates P'.

We prove that if two processes are bisimilar than they are trace equivalent. We will then show commutativity of interleaving w.r.t. bisimilarity. This implies the trace equivalence of interleaving. It turns out that proofs of bisimilarity are much more straight forward than proofs of trace equivalence. The reason is that we need to deal only with a process and the processes obtained after following one external or internal choice. So we don't have to deal with a sequence of processes involved when following a trace. So by going via bisimilarity we obtain much simpler proofs of trace equivalence.

References

[1] B. Igried and A. Setzer. Programming with monadic CSP-style processes in dependent type theory. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 28–38, New York, NY, USA, 2016. ACM.

[2] B. Igried and A. Setzer. CSP-Agda. Agda library. Available at https://github.com/csetzer/cspagdaPublic, 2017.

Derivation of elimination principles from a context

Ambrus Kaposi[1], András Kovács[1], Balázs Kőműves[2], and Péter Diviánszky[1]

[1] Eötvös Loránd University, Budapest, Hungary
{akaposi|andraskovacs}@caesar.elte.hu, divipp@gmail.com
[2] Falkstenen AB
bkomuves@gmail.com

We describe a syntactic method for deriving the specification of the dependent eliminator (induction principle) from the type formation rules and constructors of an inductive type. We observe that the work on parametricity for dependent types by Bernardy et al [1] can be used for this purpose. Logical predicates give the motives and methods for the eliminator, while logical relations provide the β computation rules. The method applies to indexed inductive types, inductive inductive types, higher inductive types and the combinations of these as well.

The construction does not involve syntactic checks like strict positivity. It only derives the the type of the eliminator and the computation rules as equalities but does not validate the existence of such constants. We describe the algorithm through examples.

An inductive type as a context

The specification of an inductive type can be given as a context. The variable names are names for the types and the constructors. For example, natural numbers can be given as the context

$$\mathbb{N} : \mathsf{U},\ \ \mathsf{zero} : \mathbb{N},\ \ \mathsf{suc} : \mathbb{N} \to \mathbb{N}.$$

We only look at closed inductive types (otherwise, the specification would be a telescope). The higher inductive type of the interval is given as

$$\mathsf{I} : \mathsf{U},\ \ \mathsf{left} : \mathsf{I},\ \ \mathsf{right} : \mathsf{I},\ \ \mathsf{segment} : \mathsf{left} \equiv \mathsf{right}.$$

A fragment of the intrinsically typed syntax of type theory is given by the context

$$\begin{aligned}&\mathsf{Con} : \mathsf{U},\ \ \mathsf{Ty} : \mathsf{Con} \to \mathsf{U},\ \ \cdot : \mathsf{Con},\ \ \rhd : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con},\ \ \iota : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma,\\ &\Pi : (\Gamma : \mathsf{Con})(A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \rhd A) \to \mathsf{Ty}\,\Gamma.\end{aligned}$$

Logical predicate interpretation

The type theory in which we write the above contexts has universes à la Russel, dependent function space and identity type. We define the logical predicate interpretation for this theory à la Bernardy [1]. That is, we define an operation $-^{\mathsf{P}}$ from the syntax to the syntax which maps a context to an extended context, a type to a predicate in the extended context and a term to a witness of the predicate corresponding to its type. The following rules specify the operation. The third rule is called parametricity or abstraction theorem.

$$\frac{\Gamma \vdash}{\Gamma^{\mathsf{P}} \vdash} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma^{\mathsf{P}} \vdash A^{\mathsf{P}} : A \to \mathsf{U}} \qquad \frac{\Gamma \vdash t : A}{\Gamma^{\mathsf{P}} \vdash t^{\mathsf{P}} : A^{\mathsf{P}}\,t}$$

Contexts are doubled: the empty context stays the same, i.e. $\cdot^{\mathsf{P}} := \cdot$. For the extended context, we add a copy of the original context and witnesses of the logical predicate for each variable: $(\Gamma, x : A)^{\mathsf{P}} := \Gamma^{\mathsf{P}}, x : A, x_{\mathsf{M}} : A^{\mathsf{P}}\,x$. The witnesses are denoted by the variable with an index $_{\mathsf{M}}$.

Terms are interpreted as follows. We look up the witness for a variable from the extended context. The universe is interpreted as predicate space, function type as the predicate which says that the function preserves predicates. The predicate for equality is equality of witnesses over the original equality. The interpretation of the eliminator J is omitted for sake of brevity — it can be given by repeated usage of J itself.

$$\begin{array}{llll}
x^{\mathsf{P}} & := x_{\mathsf{M}} & (\lambda x.t)^{\mathsf{P}} & := \lambda x\, x_{\mathsf{M}}.t^{\mathsf{P}} \\
\mathsf{U}^{\mathsf{P}} & := \lambda A.A \to \mathsf{U} & (f\,a)^{\mathsf{P}} & := f^{\mathsf{P}}\,a\,a^{\mathsf{P}} \\
((x : A) \to B)^{\mathsf{P}} & := \lambda f.(x : A)(x_{\mathsf{M}} : A^{\mathsf{P}}\,x) \to B^{\mathsf{P}}\,(f\,x) & \mathsf{refl}^{\mathsf{P}} & := \mathsf{refl} \\
(a \equiv_A a')^{\mathsf{P}} & := \lambda w.\mathsf{transport}\,A^{\mathsf{P}}\,w\,a^{\mathsf{P}} \equiv_{A^{\mathsf{P}}\,a'} a'^{\mathsf{P}} & &
\end{array}$$

A binary version $-^{\mathsf{R}}$ can be defined in an analogous way. $-^{\mathsf{R}}$ triples contexts, it produces two copies of the original context and witnesses of logical relations pointwise.

Deriving the specification of the eliminator

Applying the operator $-^{\mathsf{P}}$ on a context extends it with the motives and methods for the eliminator. For the first two examples above, we get the following additional elements:

$$\mathbb{N}_{\mathsf{M}} : \mathbb{N} \to \mathsf{U},\ \mathsf{zero}_{\mathsf{M}} : \mathbb{N}_{\mathsf{M}}\,\mathsf{zero},\ \mathsf{suc}_{\mathsf{M}} : (n : \mathbb{N})(n_{\mathsf{M}} : \mathbb{N}_{\mathsf{M}}\,n) \to \mathbb{N}_{\mathsf{M}}\,(\mathsf{suc}\,n)$$
$$\mathsf{I}_{\mathsf{M}} : \mathsf{I} \to \mathsf{U},\ \mathsf{left}_{\mathsf{M}} : \mathsf{I}_{\mathsf{M}}\,\mathsf{left},\ \mathsf{right}_{\mathsf{M}} : \mathsf{I}_{\mathsf{M}}\,\mathsf{right},\ \mathsf{segment}_{\mathsf{M}} : \mathsf{transport}\,\mathsf{I}_{\mathsf{M}}\,\mathsf{segment}\,\mathsf{left}_{\mathsf{M}} \equiv \mathsf{right}_{\mathsf{M}}$$

The computation rules can be derived using the binary logical relation interpretation $-^{\mathsf{R}}$. For example, the relation $\mathbb{N}_{\mathsf{M}}$ will have type $\mathbb{N}_0 \to \mathbb{N}_1 \to \mathsf{U}$ (instead of $\mathbb{N} \to \mathsf{U}$) where $\mathbb{N}_0$ and $\mathbb{N}_1$ are the two copies of $\mathbb{N}$. If we substitute $\mathbb{N}_{\mathsf{M}}$ for a graph of a function $f_{\mathbb{N}} : \mathbb{N}_0 \to \mathbb{N}_1$ in the whole extended context, we get the notion of a homomorphism between the natural number algebras $(\mathbb{N}_0, \mathsf{zero}_0, \mathsf{suc}_0)$ and $(\mathbb{N}_1, \mathsf{zero}_1, \mathsf{suc}_1)$.

$$\mathbb{N}_0 : \mathsf{U},\ \mathbb{N}_1 : \mathsf{U},\ f_{\mathbb{N}} : \mathbb{N}_0 \to \mathbb{N}_1,\ \mathsf{zero}_0 : \mathbb{N}_0,\ \mathsf{zero}_1 : \mathbb{N}_1,\ \mathsf{zero}_{\mathsf{M}} : f_{\mathbb{N}}\,\mathsf{zero}_0 \equiv \mathsf{zero}_1,\ \mathsf{suc}_0 : \mathbb{N}_0 \to \mathbb{N}_0,$$
$$\mathsf{suc}_1 : \mathbb{N}_1 \to \mathbb{N}_1,\ \mathsf{suc}_{\mathsf{M}} : (n_0 : \mathbb{N}_0)(n_1 : \mathbb{N}_1)(n_{\mathsf{M}} : f_{\mathbb{N}}\,n_0 \equiv n_1) \to f_{\mathbb{N}}\,(\mathsf{suc}_0\,n_0) \equiv (\mathsf{suc}_1\,n_1)$$

With a singleton contraction operation the type of $\mathsf{suc}_{\mathsf{M}}$ becomes $(n_0 : \mathbb{N}_0) \to f_{\mathbb{N}}\,(\mathsf{suc}_0\,n_0) \equiv (\mathsf{suc}_1\,(f_{\mathbb{N}}\,n_0))$ which is exactly the computation rule of the nondependent eliminator.

The computation rules for the dependent eliminator can be given by a “dependent” variant of the binary logical relation interpretation. Here the relation for $\mathbb{N}$ has type $(n : \mathbb{N}) \to \mathbb{N}_{\mathsf{M}}\,n \to \mathsf{U}$.

Further steps

If we restrict the contexts to strictly positive ones (this can be achieved using an empty universe and a restricted function space), we can define an operation which gives the type of the eliminator and the computation rules directly. We are currently working on the definition of this operator. We are also planning to extend the approach to coinductive types and to define a type theory with support for (higher) (co)inductive types using this approach.

References

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

A Type-Theoretic Alternative to LISP [*]

G. A. Kavvos[1]

Department of Computer Science, University of Oxford, United Kingdom
alex.kavvos@cs.ox.ac.uk

Abstract

To be intensional is to be finer than some predetermined extensional equality. This distinction is nowhere more pertinent than in programming: we can consider functional programs up to the function they compute, or up to their source code. A type-theoretic view of this distinction leads to a typed lambda calculus for intensional computation, and a novel categorical semantics of intensionality.

1 Introduction

In the realm of *functional computation*, we can immediately distinguish two paradigms:

- **The Extensional Paradigm**. In a purely functional world, a higher-order program can use a functional argument *extensionally*, by evaluating it at a finite number of points (a form of continuity).
- **The Intensional Paradigm**. A program can compute with the *source code*—or *intension*—of another program as an argument. It can edit, optimize, call, or simulate the running of this code.

Whereas the first paradigm led to a successful research programme on the *semantics of programming languages*, the second is often reduced to *symbolic evaluation*. But the question remains: what can the intensional paradigm contribute to programming?

2 Coping with intensionality

Intensionality involves disregarding some form of extensional equality. As a result, intensional constructions often lead to impossibility theorems or paradoxes. As an example, *quoting* is impossible. But other intensional operations are not; e.g. there are λ-terms **gnum**, **app** and **E** such that

$$\mathbf{gnum}\,\ulcorner M \urcorner =_\beta \ulcorner\ulcorner M \urcorner\urcorner \quad , \quad \mathbf{app}\,\ulcorner M \urcorner \ulcorner N \urcorner =_\beta \ulcorner M\, N \urcorner \quad , \quad \mathbf{E}\,\ulcorner M \urcorner =_\beta M$$

So intensional operations are admissible; one can go from intension ($\ulcorner M \urcorner$) to extension M; but *extension must not flow into intension.*

2.1 Enter types

Strangely, intensionality follows a *modal* typing discipline: we can read $\Box A$ as 'code of type A.' Thus, if for each $M : A$, we have $\ulcorner M \urcorner : \Box A$, then the above operations are typable:

$$\mathbf{gnum} : \Box A \to \Box\Box A \quad , \quad \mathbf{app} : \Box(A \to B) \to \Box A \to \Box B \quad , \quad \mathbf{E} : \Box A \to A$$

These types correspond to the **4**, K, and T axioms of modal logic. This observation is due to Neil Jones, but the exact connection to $\mathsf{S4}$ was drawn by Davies and Pfenning [3].

*This abstract summarises my doctoral work, which is supported by the EPSRC.

2.2 Intensional Recursion and Löb's rule

There are two ways to define a function by recursion [4]. The $\mathbf{Y}$ combinator of PCF corresponds to the *First Recursion Theorem (FRT)*. But there is also the *Second Recursion Theorem (SRT)*: for all $f \in \Lambda$ there exists $u \in \Lambda$ such that $u = f \ulcorner u \urcorner$. The SRT affords additional programming power: u has 'access' to its own source code.

Type-theoretically, the SRT proves that from $\Box A \to A$ we can infer $\Box A$. This is a variant of *Löb's rule* from provability logic [2].

2.3 Intensional PCF

To accommodate the above intuitions we introduce a new typed λ-calculus, *Intensional PCF*, by extending the dual-context S4 system of Davies and Pfenning with (a) intensional (non-functional) operations at modal types, and (b) intensional recursion. There is a caveat: intensional operations can only 'reduce' when the argument is closed. We have that [5]:

Theorem 1. *Intensional PCF is confluent, hence consistent.*

3 Categorical Semantics

All is well with the above, apart from the fact that the known categorical semantics for S4 [1] do not suit the new system. We do not want reductions $\mathsf{box}\, M \to \mathsf{box}\, N$ under a $\mathsf{box}\,(-)$ construct, yet $f = g$ implies $Ff = Fg$ for an endofunctor F. The solution is the following:

1. We replace categories with *P-categories*: the axioms of a category only hold up to a PER $\sim_{A,B}$ on each hom-set $\mathcal{C}(A, B)$. This allows the morphisms to be intensional, with the PER 'externally' specifying the extensional equality.
2. We replace cartesian comonads with *exposures*. Exposures are almost functors: they are maps on objects and morphisms that *do not respect the PERs, but reflect them instead.* Otherwise, they preserve composition and identities.

This leads to a compelling theory for the *semantics of intensionality* [6]:

Theorem 2. *There are natural examples of exposures, both from* classical logic*—corresponding to Gödel numberings—and from* realizability theory*—where they* expose *the 'implementation.'*

References

[1] Gavin M. Bierman and Valeria de Paiva. On an Intuitionistic Modal Logic. *Studia Logica*, 65(3):383–416, 2000.

[2] George S. Boolos. *The Logic of Provability.* Cambridge University Press, Cambridge, 1994.

[3] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[4] G. A. Kavvos. Kleene's Two Kinds of Recursion. *CoRR*, 2016.

[5] G. A. Kavvos. Intensionality, Intensional Recursion, and the Gödel-Löb axiom. In *Proceedings of 7th Workshop on Intuitionistic Modal Logic and Applications (IMLA 2017)*, 2017.

[6] G. A. Kavvos. On the Semantics of Intensionality. In Javier Esparza and Andrzej S. Murawski, editors, *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 10203 of *Lecture Notes in Computer Science*, pages 550–566. Springer-Verlag Berlin Heidelberg, 2017.

Quantitative Types for the Lambda-Mu Calculus with Implicit or Explicit Operations

Delia Kesner, Pierre Vial
IRIF (CNRS, and Université Paris-Diderot)

A few years after Griffin [1] observed that Feilleisen's C operator can be typed with the double-negation elimination, Parigot [2] made a major step in extending the Curry-Howard from intuitionistic to classical logic by proposing the λ_μ-calculus as a simple term notation for classical natural deduction proofs. Other calculi were proposed since then, as for example Curien-Herbelin's $\lambda\mu\tilde{\mu}$-calculus [3] based on classical sequent calculus.

Simple types are known to be unable to type some normalizing term, for instance the normal form $\Delta = \lambda x.xx$. *Intersection* types, pioneered by Coppo and Dezani [4, 5], extend simple types by resorting to a new constructor $\cap$ for types, allowing to assign a type of the form $((\sigma \to \sigma)\cap\sigma) \to \sigma$ to the term Δ. The intuition behind a term t of type $\tau_1\cap\tau_2$ is that t has both types τ_1 and τ_2. The intersection operator $\cap$ is to be understood as *idempotent* ($\sigma\cap\sigma = \sigma$), *commutative* ($\sigma\cap\tau = \tau\cap\sigma$), and *associative* ($(\sigma\cap\tau)\cap\delta = \sigma\cap(\tau\cap\delta)$). Among other applications, intersection types have been used as a *behavioural* tool to reason about several operational and semantical properties of programming languages. For example, a λ-term/program t is strongly normalising/terminating if and only if t can be assigned a type in an appropriate intersection type assignment system.

This technology turns out to be a powerful tool to reason about *qualitative* properties of programs, but not for *quantitative* ones. Indeed, we know that a term t is typable if and only if it is *e.g.* head normalizing, but we do not have any information about the number of head-reduction steps needed to head-normalize t, because of idempotency. In constrast, after the pioneering works of Gardner [6] and Kfoury [7], D. de Carvalho [8] established in his PhD thesis a relation between the size of a typing derivation in a non-idempotent intersection type system for the lambda-calculus and the head/weak-normalisation execution time of head/weak-normalising lambda-terms, respectively. Non-idempotent types have recently received a lot of attention in the domain of semantics of programming languages from a quantitative perspective (see for example [9]).

The case of the λ_μ-calculus: Different qualitative and quantitative models for classical calculi were proposed in [10, 11, 12, 13], where the characterization of operational properties was studied w.r.t. head-normalization. Intersection and union types were also studied in the framework of classical logic [14, 15, 16, 17], but no work adresses the problem from a quantitative perspective. Type-theoretical characterization of strong-normalization for classical calculi were provided both for λ_μ [18] and $\lambda\mu\tilde{\mu}$-calculus [17], but the (idempotent) typing systems do not allow to construct decreasing measures for reduction, thus a resource aware semantics cannot be extracted from those interpretations. Different small step semantics for classical calculi were developed in the framework of neededness [19, 20], without resorting to any resource aware semantical argument.

Our first contribution is the definition of a resource aware type system for the λ_μ-calculus based on non-idempotent *intersection* and *union* types. The non-idempotent approach provides very simple combinatorial arguments, only based on a decreasing *measure*, to characterize strongly normalizing terms by means of typability. In the well-known case of the λ-calculus, the measure $\mathtt{sz}\,(\Pi)$ of a derivation Π is simply given by the number of its nodes. This approach cannot be straightforwardly adapted to λ_μ, and we need now to take into account the structure (*multiplicity* and *size*) of certain types appearing in the types derivations.

The second contribution of our work is the definition of a new resource aware operational semantics for λ_μ, called $\lambda_{\mu\mathtt{r}}$, inspired from the *substitution at a distance paradigm* [21], where the reduction rules do not propagate the cuts w.r.t. the constructors of terms. The resulting calculus is compatible with the non-idempotent typing system defined for λ_μ. We then extend the typing system for λ_μ, so that the extended reduction system $\lambda_{\mu\mathtt{r}}$ preserves and decreases the size of typing derivations. We generalize the type-theoretical characterization of Strong Normalization to this new resource aware classical calculus.

References

[1] T. Griffin. A formulae-as-types notion of control. In *POPL*, pp. 47–58. ACM Press, 1990.

[2] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *LPAR*, LNCS 624, pp. 190–201. Springer, 1992.

[3] P. L. Curien and H. Herbelin. The duality of computation. In *ICFP*, pp. 233–243. ACM, 2000.

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

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

[6] P. Gardner. Discovering needed reductions using type theory. In *TACS*, LNCS 789, pp. 555–574. Springer, 1994.

[7] A. Kfoury. A linearization of the lambda-calculus and consequences. Technical report, Boston Universsity, 1996.

[8] D. de Carvalho. *Sémantiques de la logique linéaire et temps de calcul.* These de doctorat, Université Aix-Marseille II, 2007.

[9] A. Bernadet and S. Lengrand. Non-idempotent intersection types and strong normalisation. *LMCS*, 9(4), 2013.

[10] P. Selinger. *Control categories and duality: on the categorical semantics of the lambda-mu calculus.* MSCS, 11(2):207–260, 2001.

[11] S. van Bakel, F. Barbanera, and U. de'Liguoro. A filter model for the $\lambda\mu$-calculus - (extended abstract). In *TLCA*, LNCS 6690, pp. 213–228. Springer, 2011.

[12] L. Vaux. Convolution lambda-bar-mu-calculus. In *TLCA*, LNCS 4583, pp. 381–395. Springer, 2007.

[13] S. Amini and T. Erhard. On Classical PCF, Linear Logic and the MIX Rule. In *CSL*, LIPIcs 41, pp. 582–596. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

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

[15] S. van Bakel. Sound and complete typing for lambda-mu. In *ITRS*, EPTCS 45, pp. 31–44, 2010.

[16] K. Kikuchi and T. Sakurai. A translation of intersection and union types for the $\lambda\mu$-calculus. In *APLAS*, LNCS 8858, pp. 120–139. Springer, 2014.

[17] D. J. Dougherty, S. Ghilezan, and P. Lescanne. Characterizing strong normalization in the Curien-Herbelin symmetric lambda calculus: Extending the Coppo-Dezani heritage. *TCS*, 398(1-3):114–128, 2008.

[18] S. van Bakel, F. Barbanera, and U. de'Liguoro. Characterisation of strongly normalising lambda-mu-terms. In *ITRS, 2012*, EPTCS 121, pp. 1–17, 2013.

[19] Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In *TLCA*, LNCS 6690, pp. 27-44. Springer, 2011.

[20] P. Pédrot and A. Saurin. Classical by-need. In *ESOP*, LNCS 9632, pp. 616–643. Springer, 2016.

[21] B. Accattoli and D. Kesner. The structural *lambda*-calculus. In *CSL*, LNCS 6247, pp. 381–395. Springer, 2010.

Theory and Demo of PML$_2$: Proving Programs in ML

Rodolphe Lepigre

LAMA, UMR 5127 - CNRS, Université Savoie Mont Blanc, France

In the last thirty years, significant progress has been made in the application of type theory to computer languages. The Curry-Howard correspondence has been explored in two main directions. On the one hand, proof assistants like Coq or Agda are based on very expressive logics. To prove their consistency, the underlying programming languages need to be restricted to contain only programs that can be proved terminating. As a result, they forbid the most general forms of recursion. On the other hand, functional programming languages are well-suited for programming, as they impose no restriction on recursion. However, they are based on inconsistent logics (i.e. the empty type is inhabited).

The aim of PML$_2$ is to provide a uniform environment in which programs can be designed, specified and proved. The idea is to combine a full-fledged, ML-like programming language with an enriched type-system allowing the specification of computational behaviours. This language can thus be used as ML for programming, or as a proof assistant for proving properties of ML programs. As there is no distinction between programs and proofs, programming and proving constructs can be mixed. For instance, proofs can be composed with programs for them to transport properties (e.g. addition carrying its commutativity proof). In programs, proof mechanisms can be used to eliminate unreachable code.

Examples. To illustrate the proof mechanism, we will consider simple examples of proofs on unary natural number. Their type is given bellow, together with the corresponding addition function defined using recursion on its first argument.

```
type rec nat = [Zero ; Succ of nat]
val rec add : nat ⇒ nat ⇒ nat = fun n m → case n of
  | Zero[_] → m
  | Succ[k] → Succ[add k m]
```

As a first example, we will show that `add Zero n` $\equiv$ `n` for all `n`.[1] To express this property we can use the type `∀n:ι, add Zero n ≡ n`, where ι can be read as the set of all values. This statement is proved below, but its symmetric does not hold.

```
val add_z_n : ∀n:ι, add Zero n ≡ n = {}
// val fails : ∀n:ι, add n Zero ≡ n = {}
```

The first proof is immediate since `add Zero n` reduces to `n` by definition of `add`. The second proof fails as we can find `n` such that `add n Zero` is not equivalent to `n`. The value `False` is a suitable counterexample since the computation of `add False Zero` produces a runtime error. Indeed, `False` is not matched in the case analysis that is used in the definition of `add`. We can however prove a weaker property using the type `∀n∈nat, add n Zero ≡ n`, which corresponds to a (dependent) function taking as input a number `n` and returning a proof of `add n Zero ≡ n`. This property can be proved using induction and case analysis as follows.

```
val rec add_n_z : ∀n∈nat, add n Zero ≡ n = fun n → case n of
  | Zero[_] → {}
  | Succ[k] → let ih = add_n_z k in {}
```

[1] The ($\equiv$) relation can be thought of as a form of observational equivalence.

If `n` is `Zero`, then we need to show `add Zero Zero` $\equiv$ `Zero`, which is immediate by definition of `add`. In the case where `n` is `Succ[k]` we need to show `add Succ[k] Zero` $\equiv$ `Succ[k]`. By definition of `add`, this reduces to `Succ[add k Zero]` $\equiv$ `Succ[k]`. We can hence use the induction hypothesis `add_n_z k` to learn `add k Zero` $\equiv$ `k` and conclude the proof. Thanks to one more lemma we easily obtain the commutativity of the `add` function.

```
val rec add_n_s : ∀n m∈nat, add n Succ[m] ≡ Succ[add n m] = fun n m →
  case n of | Zero[_] → {}
            | Succ[k] → let ind_hyp = add_n_s k m in {}

val rec add_comm : ∀n m∈nat, add n m ≡ add m n = fun n m →
  case n of | Zero[_] → let lem = add_n_z m in {}
            | Succ[k] → let ih  = add_comm k m in
                        let lem = add_n_s m k in {}
```

It is important to note that, in our system, a program that is considered as a proof needs to go through a termination checker. It is however easy to see all the proofs given here are terminating, and hence valid.

Many more examples of proofs (and programs) have been written using our implementation of the system, which should be made publicly available in a few weeks. They include (but are not limited to) proofs on unary natural numbers (involving addition and product), lists, vectors of a given size (as a subtype of lists) and sorting algorithms.

Realisability model and semantics. The theoretical foundations of PML$_2$ have been studied by the author in his PhD thesis [3]. Most of the ideas contained in this work are based on the exploratory work of Christophe Raffalli on PML [5]. Although this first version of the system was encouraging on the practical side, it did not stand on solid theoretical grounds. The semantics of PML$_2$ is based on a call-by-value classical realisability model first presented in [2]. This theoretical framework allows for a relaxation of value restriction, which is essential for working with dependent function types in the presence of effects. Dependent functions are an important component of the system, as they allows a form of typed quantification.

Another important part of the semantics of PML$_2$ is a specific notion of subtyping introduced in a joint work of Christophe Raffalli and the author [4]. This framework yields a type system that can be directly implemented, using (mostly) syntax-directed typing and subtyping rules. In particular, it provides a very general notion of infinite proofs, which well-foundedness must be established using an external solver based on the size-change principle [1].

References.

[1] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92. ACM, 2001.

[2] Rodolphe Lepigre. A Classical Realizability Model for a Semantical Value Restriction. In *ESOP*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.

[3] Rodolphe Lepigre. *Semantics and Implementation of an Extension of ML for Proving Programs.* PhD thesis, Université Grenoble Alpes, 2017.

[4] Rodolphe Lepigre and Christophe Raffalli. Practical Subtyping for System F with Sized (Co)-Induction. Prototype implementation: `http://lama.univ-savoie.fr/subml/`, 2016.

[5] Christophe Raffalli. PML: a new proof assistant. Prototype implementation: `http://lama.univ-savoie.fr/~raffalli/pml`, Talk at the Types workshop, 2007.

Intersection and Union Types from a Proof-functional point of View[*]

Luigi Liquori and Claude Stolze

Université Côte d'Azur, Inria, France

This talk is a contribution to the study of types with *intersection* and *union* and the role of such type systems in *logical* investigations. The talk inspects (i) the relationship between pure (Curry-style) and typed (Church-style) lambda-calculi and their corresponding proof-functional logics as dictated by the well-known Curry-Howard correspondence, (ii) the algorithmic issues related with subtyping, and (iii) the possibility of including in theorem provers proof-functional connectives like type intersection and union.

Proof-functional logical connectives allow reasoning about the structure of logical proofs, in this way giving to the latter the status of *first-class* objects. This is in contrast to classical *truth-functional* connectives where the meaning of a compound formula is dependent only on the truth value of its subformulas.

Proof-functional connectives represent evidences as "polymorphic" constructions, that is, a *same* evidence can be used as a proof for different sentences. Pottinger [14] firstly introduced a conjunction, called *strong conjunction* $\cap$ (as known as *type intersection*), requiring more than the existence of constructions proving the left and the right hand side of the conjuncts.

According to Pottinger: *"The intuitive meaning of $\cap$ can be explained by saying that to assert $A \cap B$ is to assert that one has a reason for asserting A which is also a reason for asserting B"*. This interpretation makes inhabitants of $A \cap B$ as polymorphic evidences for both A and B. Later, Lopez-Escobar [11] presented the first proof-functional logic with strong conjunction as a special case of ordinary conjunction. Mints [12] presented a logical interpretation of strong conjunction using *realizers*: the logical predicate $r_{A\cap B}[M]$ is true if the pure lambda-term M is a realizer for both the formula $r_A[M]$ *and* $r_B[M]$.

[10, 6] presented Church-style versions of the type assignment systems of Barendregt-Coppo-Dezani [3] and Barbanera-Dezani-de'Liguoro [2], adding to type intersection another proof-functional operator, the *strong disjunction* $\cup$ (as known as *type union*). Paraphrasing Pottinger's point of view, we could say that the intuitive meaning of $\cup$ can be explained by saying that if we have a reason to assert A (or B), then the same reason will also assert $A \cup B$. This interpretation makes inhabitants of $(A \cup B) \supset C$ as polymorphic evidences for both $A \supset C$ and $B \supset C$. In [5] we extended Mints' logical interpretation of type intersection with type union. Symmetrically to intersection, the logical predicate $r_{A\cup B}[M]$ succeeds if the pure lambda-term M is a realizer for either the formula $r_A[M]$ *or* $r_B[M]$.

Subtyping Algorithm. Roger Hindley gave first a subtyping algorithm for type intersection [9], and there is a rich literature reducing the subsetting problem in presence of set intersection and set union to set constraint-based problem: good references are [1, 4, 7]. We present a decidable algorithm for subtyping $\mathcal{A}$ in presence of type intersection and union: the algorithm is conceived to work for the minimal type theory Ξ (ie. axioms 1 to 14, as presented in [2]). The algorithm $\mathcal{A}$ is proved to be sound and complete.

[*]Work supported by the COST Action CA15123 EUTYPES "The European research network on types for programming and verification".

Dependent Types / Logical Frameworks. All of the work on understanding the logical aspects of intersection, union, and subtyping took place in the Curry-style framework. This was natural given the fact that type assignment was the most natural framework for type intersection and union, because the typing rules are not *syntax directed*. But the fact that the Curry-Howard correspondence requires explicitly-typed terms poses a compelling question: *can a logical investigation of type intersection and union in presence of subtyping, take place in the context of an explicitly-typed lambda-calculus?* In the literature, Frank Pfenning work on *Refinement Types* [13] pioneered an extension of Edinburgh Logical Framework (LF) [8] with subtyping and type intersection. Our aim is to study extensions of LF featuring fully fledged proof-functional logical connectives like strong conjunction, strong disjunction in presence of subtyping and *relevant implication*.

The motivation is that success here should point the way towards applications of type intersection and union in logical frameworks. Studying the behavior of proof-functional connectives would be beneficial to existing interactive theorem provers such as Coq or Isabelle, and dependently typed programming languages such as Agda, Beluga, Epigram, or Idris, just to mention a few. The hope is that they can provide as much insight into logical systems as they have in the computational arena.

Prototype Implementation. We are current implementing a small kernel for a logical framework prototype featuring type union and intersection. The actual type system features an experimental implementation of dependent-types à la LF, and of a primitive *Read-Eval-Print-Loop* (REPL). We are currently trying to integrate our algorithm $\mathcal{A}$ to the type checker engine. The actual state of the prototype can be retrieved at `https://github.com/cstolze/Bull`.

References

[1] A. Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2):79–111, 1999.

[2] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Intersection and union types: syntax and semantics. *Inf. Comput.*, 119(2):202–230, 1995.

[3] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[4] F. Damm. Subtyping with union types, intersection types and recursive types II. RR RR-2259, INRIA, 1994.

[5] D. J. Dougherty, U. de'Liguoro, L. Liquori, and C. Stolze. A realizability interpretation for intersection and union types. In *APLAS*, volume 10017 of *LNCS*, pages 187–205, 2016.

[6] D. J. Dougherty and L. Liquori. Logic and computation in a lambda calculus with intersection and union types. In *LPAR*, volume 6355 of *LNCS*, pages 173–191, 2010.

[7] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *POPL*, pages 281–292, 2004.

[8] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[9] J. R. Hindley. The simple semantics for coppe-dezani-sallé types. In *International Symposium on Programming*, pages 212–226, 1982.

[10] L. Liquori and S. Ronchi Della Rocca. Intersection typed system *à la* Church. *Information and Computation*, 9(205):1371–1386, 2007.

[11] E. G. K. Lopez-Escobar. Proof functional connectives. In *Methods in Mathematical Logic*, volume 1130 of *LNCS*, pages 208–221, 1985.

[12] G. Mints. The completeness of provable realizability. *Notre Dame Journal of Formal Logic*, 30(3):420–441, 1989.

[13] F. Pfenning. Refinement types for logical frameworks. In *TYPES*, pages 285–299, 1993.

[14] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.

Definitional Extensions in Type Theory Revisited

Georgiana Elena Lungu[1] and Zhaohui Luo[2]

[1] Royal Holloway, Univ of London
Georgiana.Lungu.2013@live.rhul.ac.uk
[2] Royal Holloway, Univ of London
zhaohui.luo@hotmail.co.uk

The notion of definitionality or extension by definition was first formulated by Kleene [2] for first order theories. In first order logic we consider an extension obtained by adding a new symbol and an axiom that a formula involving the new symbol is equivalent to one in the base system. For an extension, conservativity means that all the formulas that are syntactically in the base system hold in the extension only if they hold in the base system. Definitionality, in addition, means that, an encompassing formula holds if and only if the corresponding formula holds if it is obtained by replacing the occurrences of the new formula with the equivalent ones from the base system.

We are interested in a similar notion for type theories like Martin Löf's type theory [6] or UTT [4]. We consider two different notions of definitional extension and, in particular, reformulate them in new ways, leading to a more generic notion of definitional extension that covers both as special cases. In the following, we use LF to denote Martin-Löf's logical framework with labelled lambda-expressions (see Chapter 9 of [4]).

The first notion of definitional extension is related to the notion of conservativity as studied in [1, 3] where an embedding of a type theory into its extension is used and induces a particular notion of definitional extension with new symbols. In the setting of LF, it can be formulated as follows. (1) Let T and T' be type theories specified in LF and T' an extension of T by adding new terms and rules. (2) Let f be a mapping from the terms of T' to those of T such that (i) $f|_T = id_T$, and (ii) the new rules involving the new terms in T' all become admissible under f in T. Then T' is a definitional extension of T iff T' is a conservative extension of T and the definition rules of the form $\frac{\Gamma\vdash k:K \quad \Gamma\vdash f(k):K}{\Gamma\vdash k=f(k):K}$ are admissible in T'.[1] For example, we may consider Σ, a type theory with Σ-types, and $\Sigma[\times]$, the extension of Σ with product types with expected rules. A syntactic map can be defined to map product types $A \times B$ to the Σ-type $\Sigma(A, [x{:}A]B)$ (and similarly for pairs and projections). Then, if in $\Sigma[\times]$ the definition rules such as $\frac{\Gamma\vdash A:Type \quad \Gamma\vdash B:Type}{\Gamma\vdash A\times B=\Sigma(A,[x:A]B)}$ are admissible, then $\Sigma[\times]$ is a definitional extension of Σ. In this setting, some meta-theoretic properties are carried over from T to its definitional extension T'. For instance, if kind uniqueness holds for T, so does it for T'.

The above notion of definitional extension relies on the existence of a syntactic mapping and it does not cover some more general situations where, for example, the set of terms of the extension is the same as that of the extended calculus. It also does not deal with situations where new judgement forms are added. Coercive subtyping is such an example, with both of these features. It has been discussed by Xue et al. [7, 5, 8]. Instead of considering mapping between terms, they have described a definitional extension by mapping a derivation tree in the extension into one in the original calculus such that their conclusions are definitionally equal.[2]

[1] Another way to think of this is that T' extends T with new terms and new rules, including those definition rules which correspond to the definition axiom in Kleene's setting of first-order theories.

[2] Note that Xue in [7] does not consider situations with new forms of judgements. Rather, he considers the calculus $T[C]_{0K}$ which has the same judgement forms with the whole calculus and is a conservative extension of the original calculus.

Here we propose a more general notion of definitional extension that has both notions above as special cases. Let T' be an extension of T with new terms, new rules and/or new forms of judgements. We define a notion of replacement of a judgement J' in T' by a judgement J in T based on a mapping $m : \mathcal{J}_{T'} \longrightarrow \mathcal{J}_T$ from T'-judgements to T-judgements so that m is the identity when restricted to T-judgements ($m|_{\mathcal{J}_T} = id_{\mathcal{J}_T}$) and, when restricted to derivable judgements, respects definitional equality. Note that the mapping m is now based on judgements, not on terms. This allows us to deal with the situation where the terms in T and those in T' are the same (as in coercive subtyping). For the first notion of definitional extension given above based on a syntactic mapping f, we can simply take m to be the extension of f to judgements. For the notion of definitional extension for coercive subtyping, as discussed in [5, 8], m maps a judgement $\Gamma \vdash A \leq_c B$ to $\Gamma \vdash c : (A)B$. It is important to note that the mapping m is syntactic and does not preserve derivability and, because of this, the definition of replacement is more complex and subtle in order to introduce and consider only derivable judgements (and we omit the details in the current abstract).

For an extension to be definitional, we require that it be conservative, and that any judgement in the extension have a replacement in the base calculus w.r.t. any of its derivation trees. This definition is similar to the one given in [7], in first place because even though a replacement is defined w.r.t. to a derivation, this definition refers to judgements as opposed to derivations, and it can cover new forms of judgements.

Note that, in order to discuss definitionality, it is important to understand what to replacement means, what a judgement can be replaced with, and how replacement should be done in a more general setting where, for example, there are new forms of judgements. In the previous settings, the answer to these questions was essentially covered by definition rules. Intuitively, one simply replaces a judgement with a definitionally equal one in a derivation tree to obtain a valid derivation tree (possibly with the addition to some more equality judgements). This cannot be the case for new forms of judgements or even for those extensions which add new forms of entries to the assumptions. These situation are all covered with our notion of definitional extenstion.

References

[1] M. Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, Univ of Edinburgh, 1995.

[2] S. Kleene. *Introduction to Metamathematics.* North Holland, 1952.

[3] P. LeFanu Lumsdaine. *Higher Categories from Type Theories.* PhD thesis, CMU, 2010.

[4] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* OUP, 1994.

[5] Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: theory and implementation. *Information and Computation*, 223:18–42, 2013.

[6] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[7] T. Xue. Definitional extension in type theory. In R. Matthes and A. Schubert, editors, *LIPIcs Proceedings 19th International Conference on Types for Proofs and Programs*, 2013.

[8] T. Xue. *Theory and Implementation of Coercive Subtyping.* PhD thesis, Royal Holloway, 2013.

Adjective Clustering in the Mizar Type System

Adam Naumowicz *

Institute of Informatics, University of Bialystok, Poland

1 Introduction

The fundamental properties and basic terminology of the soft type system currently implemented in the Mizar proof checking system [1, 4] was presented at TYPES 2016, see [8]. There had been some notable attempts to translate the underlying concepts into other proof systems, namely HOL Light [6] and more recently Isabelle [5]. A complete translation preserving the notion of obviousness of the Mizar checker would require a thorough treatment of the type system, including all its features.

A particular feature of the type system is the use of "adjectives" or "attributes" as constructors of flexible type hierarchies in the collection of interdependent Mizar articles forming the Mizar Mathematical Library (MML). In principle, Mizar adjectives are semantically variants of (dependent) predicates. But their natural language based syntactic form, together with the built-in type inference automation make them quite a powerful formalization mechanism which seems indispensable for most non-trivial Mizar formalizations (cf. [9]). Our goal is to present the adjective-based Mizar mechanisms in more detail to facilitate a wider understanding of Mizar formalizations and experimenting with encoding the contents of MML in other formalisms and/or frameworks.

2 Clustering

Let us briefly recall here that a "cluster of adjectives" in the Mizar jargon means a collection of attributes (constructors of adjectives) with boolean values associated with them (negated or not) and their arguments. The tree-like hierarchical structure of Mizar types is built by the widening relation which uses such collections of adjectives to extend existing types [2]. Grouping adjectives in clusters enables automation of some type inference rules. Such rules are encoded in the form of so called registrations. Previously proved registrations can subsequently be used to secure the non-emptiness of Mizar types (existential registrations), to allow formulating and automating relationships between adjectives (conditional registrations) and to store adjectives that are always true for instantiations of terms with certain arguments (functorial registrations), cf. [3].

As originally implemented in Mizar, the role of adjective processing was mostly syntactic, i.e. the Analyzer module automatically "rounded-up" the information from all available registrations to disambiguate used constructors and check their applicability. The semantic role was restricted to processing only the type information for the terms explicitely stated in an inference. Also, attributive statements as premises or conclusions were not "rounded-up". The available automation did not take into account the potential of applying registrations to every element of a class of equal terms generated in the Equalizer module as a consequence of the equality calculus.

An optimization algorithm was later implemented [7] to enable "rounding-up" the, so called, "super clusters", i.e. clusters of adjectives collected from various representations of terms that happen to be aggregated in the same equality class as a consequence of equality processing. Now let us look at a simple example with two typical functorial registrations for integers encoded in the Mizar syntax:

*The author is supported by COST Action CA15123.

```
registration
  let i be even Integer, j be Integer;
  cluster i*j -> even;
end;
registration
  let i be even Integer, j be odd Integer;
  cluster i+j -> odd;
end;
```

With these registrations imported, Mizar's Checker module can, for example, infer automatically the following statements as obvious for any `i`, `e` and `o` being integers:

```
e is even implies i*e is even;
e is even & o is odd implies e+o is odd;
e is even & o is odd implies (i*e)+o is odd;
```

In general, the equality classes may have numerous representatives, as well as multiple types, which in turn have their arguments of the same form, and so on. As a class may have several types and several term instances that may match the same registration, the result of matching is a list of instantiations of classes for the loci used in a registration.

3 Final Remarks

Interestingly, the implementation of the above mechanism reused some of the data structures previously developed for the Unifier module, where an algebra of substitutions is used to contradict a given universal formula. The main difference is when joining instantiation lists - in the Unifier the longer substitution is absorbed, while within the "super cluster" matching algorithm the longer substitution remains.

It is also worth mentioning that in the above example all the attributes were absolute, i.e. their only argument was the subject. But in general, the subject may be defined with a type that has its own (explicit or implicit) arguments, and so the adjective has more implicit arguments. Proper and efficient "rounding-up" clusters of adjectives with many arguments that can appear in clusters several times (but possibly with different arguments) is another non-trivial issue.

References

[1] Bancerek, G. et al., Mizar: State-of-the-Art and Beyond. In M. Kerber et al. (Eds.), Intelligent Computer Mathematics, CICM 2015, LNAI 9150, 261-279, 2015.

[2] Bancerek. G., On the Structure of Mizar Types. ENTCS 85(7), 6985, 2003.

[3] Grabowski, A., Korniłowicz, A., Naumowicz, A., Mizar in a nutshell, Journal of Formalized Reasoning 3 (2) (2010) 153–245. `http://jfr.unibo.it/article/view/1980`.

[4] Grabowski, A., Korniłowicz, A., Naumowicz, A., Four Decades of Mizar - Foreword. Journal of Automated Reasoning 55(3), pp. 191-198, 2015.

[5] Kaliszyk, C., Pąk, K., Urban, J., Towards a Mizar environment for Isabelle: foundations and language. CPP 2016: 58-65.

[6] Kunčar, O., Reconstruction of the Mizar Type System in the HOL Light System. Proc. of WDS'10.

[7] Naumowicz, A., Enhanced Processing of Adjectives in Mizar. In A. Grabowski and A. Naumowicz (Eds.), Computer Reconstruction of the Body of Mathematics, Studies in Logic, Grammar and Rhetoric 18(31), 89-101, 2009.

[8] A. Naumowicz and J. Urban, A Guide to the Mizar Soft Type System, TYPES 2016 Book of abstracts, `http://www.types2016.uns.ac.rs/images/abstracts/naumowicz.pdf`.

[9] Schwarzweller, C.: Mizar Attributes: A Technique to Encode Mathematical Knowledge into Type Systems. *Studies in Logic, Grammar and Rhetoric* **10**(23) (2007) 387–400.

Parametric Quantifiers for Dependent Types

Andreas Nuyts[1], Andrea Vezzosi[2], and Dominique Devriese[1]

[1] KU Leuven, Leuven, Belgium
[2] Chalmers University of Technology, Gothenburg, Sweden

Many type systems and functional programming languages support functions that are parametrized by a type. For example, we may create a tree flattening function $\mathsf{flatten}\,\alpha : \mathsf{Tree}\,\alpha \to \mathsf{List}\,\alpha$ that works for any type α. If the implementation of a parametrized function does not inspect the particular type α that it is operating on, possibly because the type system prohibits this, then the function is said to be *parametric*: it applies the same algorithm to all types. From this knowledge, we obtain various useful 'free theorems' about the function. For example, if we have a function $f : A \to B$, then we know that $\mathsf{listmap}\,f \circ \mathsf{flatten}\,A = \mathsf{flatten}\,B \circ \mathsf{treemap}\,f$. If parametricity is enforced by the type system, as is the case in System F but also in a programming language like Haskell, then we can deduce such free theorems purely from a function's type signature, without knowledge of its implementation. This allows parts of a function's contract to be enforced by the type-checker; a powerful feature.

Existing work on parametricity in dependent type systems such as Martin-Löf Type Theory (MLTT) has been able to show that the expected parametricity results hold for functions that produce values of a small type [AGJ14, Tak01, KD]. Below, we illustrate that existing dependent type systems insufficiently enforce parametricity in the sense that some parametricity theorems do not hold where large types are involved. The central aim of this paper is to resolve this issue by equipping dependent type theory with additional parametric quantifiers.

Encoding lists in System F In order to expose the problem that occurs in dependent type theory, we will elaborate an example that shows the power of parametricity in System F, but which does not carry over to dependent type theory: the standard Church encoding of lists. Given a type B, we define the type of Church lists over B as $\mathsf{ChList}\,B = \forall\alpha.\alpha \to (B \to \alpha \to \alpha) \to \alpha$. Parametricity guarantees that elements of this type are in one-to-one correspondence with lists of elements of B. Intuitively, this can be understood as follows: values of the type $\mathsf{ChList}\,B$ have the form $\Lambda\alpha.\lambda(\mathsf{nil}' : \alpha).\lambda(\mathsf{cons}' : B \to \alpha \to \alpha).t$, where the body t has type α. The only ways to create terms of the unknown type α is by using the arguments nil' and cons', so the syntactical term t can be converted into a list by removing the primes.

Encoding lists in dependent type theory. Dependent type theory departs from System F in that it erases the strict dichotomy between types and values. In particular, types can be used as data, e.g. we can consider lists of types. The function type former $\to$ from System F, is replaced with the dependent function type former Π. If S is a type and T is a type depending on a variable $x : S$, then the type $\Pi(x : S).T$ contains functions f that map any value $s : S$ to a value $f\,s : T[s/x]$. When T does not depend on x, we have recovered the ordinary function type $S \to T$ from System F. If we disregard parametricity, we may also use Π to recover the $\forall$ type former from System F. If the domain S is a type of types $\mathcal{U}$, also called a universe, then the function type $\Pi(\alpha : \mathcal{U}).T$ corresponds to the polymorphic type $\forall\alpha.T$ from System F. So we can translate our Church encoding of lists to dependent types: $\mathsf{ChList}\,B = \Pi(X : \mathcal{U}).X \to (B \to X \to X) \to X$. But does this still encode the type of lists?

The answer is not in general positive, and an easy counterexample can be constructed if we let B be the universe $\mathcal{U}$ itself. Then the following element $\mathsf{exoticList} = \lambda X.\lambda\mathsf{nil}'.\lambda\mathsf{cons}'.\mathsf{cons}'\,X\,\mathsf{nil}' : \mathsf{ChList}\,\mathcal{U}$ is a blatant violation of how our encoding was intended to be used: given a type X and

nil' and cons' operators on X, it returns the list of length one containing X. So the argument X, a type purely provided for type-checking purposes, is used as a value in the list! As a result, exoticList does not represent a fixed list, but a list whose content depends on what type X we are eliminating to. This is definitely not something we want to allow. [1]

Contributions. We present a dependent type system **ParamDTT** in which dependencies can be either *parametric* or *continuous*. Correspondingly, we obtain relationally parametric quantifiers $\forall$ and $\exists$ alongside the usual (continuous) quantifiers Π and Σ.

We make parametricity theorems provable internally using a type former called Glue (first used by [CCHM16] in their quest for computational univalence), and its (novel) dual which we call Weld. These are an alternative for the operators by [BCM15]. Both Glue and Weld have some dependencies that are not continuous and that we cannot prove further parametricity theorems about. This is represented by a third *pointwise* modality.

We construct Church initial algebras and final co-algebras of indexed functors. We prove their universal properties (up to universe level issues) internally, which to our knowledge has not been done before in any type system. These internal proofs have some pointwise dependencies, indicating that internal parametricity does not apply again to those dependencies.

Annotating (co-)recursive types with a size bound on their elements is a modular way to enforce termination and productivity of programs. We construct initial algebras and final co-algebras of a large class of indexed functors using induction on, and parametric quantification over size bounds. We again prove their universal properties internally.

We implement an extension to the dependently typed language Agda, which type-checks ParamDTT and thus shows that its computational behaviour is sufficiently well-behaved to allow for automated type-checking. [2] We expect that ParamDTT minus its equality axioms, which block computation of the J-rule, satisfies all desired computational properties.

We prove soundness by constructing a presheaf model in terms of iterated reflexive graphs (more commonly called cubical sets), based on the reflexive graph model by [AGJ14] and enhancements by [BCM15].

Acknowledgements Andreas Nuyts and Dominique Devriese hold a Ph.D. Fellowship and a postdoctoral mandate (resp.) from the Research Foundation - Flanders (FWO).

References

[AGJ14] Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *Principles of Programming Languages*. ACM, 2014.

[BCM15] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electr. Notes Theor. Comput. Sci.*, 319:67–82, 2015.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[KD] Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In *Computer Science Logic (CSL), 2013.*

[Tak01] Izumi Takeuti. The theory of parametricity in lambda cube. 2001. Technical report, Kyoto University.

[1] When we make universe levels explicit, we have a type $\mathsf{ChList}_i\,\mathcal{U}_j$ of lists over $\mathcal{U}_j$ that eliminate to $X : \mathcal{U}_i$. The reader may object that we either cannot write exoticList ($i > j$) or we cannot use that exoticList's contents ($i \leq j$), and hence there is no problem. Such theorem is conceivable, but does not withstand the general issue that the ability to use type arguments as data, breaks parametricity theorems. E.g. in System F, all elements of $\forall\alpha.T$ (where T is closed) are constant by parametricity, but $\Pi(X : \mathcal{U}).T$ contains the identity function if $T = \mathcal{U}$.

[2] https://github.com/agda/agda/tree/parametric

Axioms for univalence

Ian Orton and Andrew Pitts

Cambridge University
Ian.Orton@cl.cam.ac.uk
Andrew.Pitts@cl.cam.ac.uk

We show that, within Martin-Löf Type Theory, the univalence axiom [4] is equivalent to function extensionality [4] and axioms (1) to (5) given in Table 1. When constructing a model satisfying univalence, experience shows that verifying these axioms is often simpler than verifying the full univalence axiom directly. We show that this is the case for cubical sets [1].

	Axiom		Premise(s)		Equality		
(1)	*unit*	:			A	$=$	$\sum_{a:A} 1$
(2)	*flip*	:			$\sum_{a:A}\sum_{b:B} C\ a\ b$	$=$	$\sum_{b:B}\sum_{a:A} C\ a\ b$
(3)	*contract*	:	*isContr* A	$\to$	A	$=$	1
(4)	*unit*β	:			*coerce unit* a	$=$	$(a, *)$
(5)	*flip*β	:			*coerce flip* (a, b, c)	$=$	(b, a, c)

Table 1: $(A, B : \mathcal{U},\ C : A \to B \to \mathcal{U},\ a : A,\ b : B$ and $c : C\,a\,b$, for some universe $\mathcal{U})$

First recall some standard definitions/results in Homotopy Type Theory (HoTT). A type A is said to be *contractible* if the type $isContr(A) :\equiv \sum_{a_0:A}\prod_{a:A}(a_0 = a)$ is inhabited, where $=$ is propositional equality. It is a standard result that singletons are contractible: for every type A and element $a : A$ the type $sing(a) :\equiv \sum_{x:A}(a = x)$ is contractible. We say that a function $f : A \to B$ is an *equivalence* if for every $b : B$ the fiber $fib_f(b) :\equiv \sum_{a:A}(f\ a = b)$ is contractible. Finally, we can define a function *coerce* $: (A = B) \to A \to B$ which, given a proof that $A = B$, will coerce values of type A into values of type B.

The axioms in Table 1 all follow from the univalence axiom. The converse is also true. The calculation on the right shows how to construct an equality between types A and B from an equivalence $f : A \to B$. This proof, and many other results described in this paper, have been formalised in the proof assistant Agda [3]. Details can be found at http://www.cl.cam.ac.uk/~rio22/agda/axi-univ.

$$\begin{aligned} A &= \sum_{a:A} 1 && \text{by (1)} \\ &= \sum_{a:A}\sum_{b:B} f\ a = b && \text{by (3) on } sing(fa) \\ &= \sum_{b:B}\sum_{a:A} f\ a = b && \text{by (2)} \\ &= \sum_{b:B} 1 && \text{by (3) on } fib_f(b) \\ &= B && \text{by (1)} \end{aligned}$$

The univalence axiom is not simply the ability to convert an equivalence into an equality, but also the fact that this operation itself forms one half of an equivalence. It can be shown (e.g. [2]) that this requirement is satisfied whenever *coerce* $(ua(f, e)) = f$ for every $(f, e) : Equiv\ A\ B$, where $ua : Equiv\ A\ B \to A = B$ is the process outlined above. In order to prove this we use axioms *unit*β and *flip*β. Had we derived *unit* and *flip* from univalence, these properties would both hold. Note that we need no assumption about *contract* since, in the presence of function extensionality, all functions between contractible types are propositionally equal.

It is easily shown that *coerce* is compositional, and so we can track the result of *coerce* at each stage to see that coercion along the composite equality $ua(f, e)$ gives us the following:

$$a \quad\mapsto\quad (a,\ *) \quad\mapsto\quad (a,\ f\ a,\ \mathit{refl}) \quad\mapsto\quad (f\ a,\ a,\ \mathit{refl}) \quad\mapsto\quad (f\ a,\ *) \quad\mapsto\quad f\ a$$

Experience shows that the first two axioms are simple to verify in many potential models of univalent type theory. To understand why, it is useful to consider the interpretation of $\mathit{Equiv}\ A\ B$ in a model of intensional type theory. Propositional equality in the type theory is not interpreted as equality in the model's metatheory, but rather as a construction on types e.g. path spaces in models of HoTT. Therefore, writing $[\![X]\!]$ for the interpretation of a type X, an equivalence in the type theory will give rise to morphisms $f : [\![A]\!] \to [\![B]\!]$ and $g : [\![B]\!] \to [\![A]\!]$ which are not exact inverses, but rather are inverses modulo the interpretation of propositional equality, e.g. the existence of a path connecting x and $g(f\,x)$. However, in many models the interpretations of A and $\sum_{a:A} 1$, and of $\sum_{a:A}\sum_{b:B} C\ a\ b$ and $\sum_{b:B}\sum_{a:A} C\ a\ b$ will be isomorphic, i.e. there will be morphisms going back and forth which are inverses up to equality in the model's metatheory. This means that we can satisfy *unit* and *flip* by proving that this stronger notion of isomorphism gives rise to a propositional equality between types.

We also assume function extensionality. Every model of univalence must satisfy function extensionality [4, Section 4.9], but it is often easier to check function extensionality than the full univalence axiom. This leaves the *contract* axiom, which captures the homotopical condition that every contractible space is equivalent to a point. The hope is that the previous axioms should come almost "for free", leaving this as the only non-trivial condition to check.

As an example, consider the cubical sets model presented in [1]. In this setting function extensionality holds trivially [1, Section 3.2]. There is a simple way to construct paths between strictly isomorphic types $\Gamma \vdash A, B$ in the presheaf semantics by defining a new type $P_{A,B}$:

$$P_{A,B}(\rho, i) :\equiv \begin{cases} A(\rho) & \text{if } i = 0 \\ B(\rho) & \text{if } i \neq 0 \end{cases} \qquad (\text{where } \rho \in \Gamma(I), i \in \mathbb{I}(I) \text{ for } I \in \mathcal{C})$$

The action of $P_{A,B}$ on morphisms is inherited from A and B, using the isomorphism where necessary. $P_{A,B}$ has a composition structure [1, Section 8.2] whenever A and B do, whose associated *coerce* function is equal to the isomophism. This construction is related to the use of a case split on $\varphi\rho = 1$ in [1, Definition 15]. Finally, given a type $\Gamma \vdash A$ and using the terminology from [1, Section 4.2], the *contract* axiom can be satisfied by taking $\Gamma, i : \mathbb{I} \vdash \mathit{contract}\ A\ i$ to be the type of partial elements of A of extent $i = 0$. The type $\mathit{contract}\ A\ i$ has a composition structure whenever A does. This construction is much simpler than the *glueing* construction that is currently used to prove univalence, and perhaps makes it clearer why the closure of cofibrant propositions under $\forall$ is required [1, Section 4.1].

References

[1] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. Preprint, December 2015.

[2] D. R. Licata. Weak univalence with "beta" implies full univalence, homotopy type theory mailing list, 2016. http://groups.google.com/d/msg/homotopytypetheory/j2KBIvDw53s/YTDK4DONFQAJ.

[3] U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

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

An Effectful Way to Eliminate Addiction to Dependence

P.-M. Pédrot[1] and N. Tabareau[2]

[1] University of Ljubljana
[2] Inria Rennes - Bretagne Atlantique

The gap between dependent type theories such as CIC and mainstream programming languages comes to a large extend from the absence of effects in type theories, because of its complex interaction with dependency. For instance, it has already been noticed that inductive types and dependent elimination do not scale well to CPS translations and classical logic [1, 3]. Furthermore, the traditional way to integrate effects from functional programming using monads does not scale to dependency because the monad leaks in the type during substitution, preventing any straightforward adaptation.

To solve this conundrum, we propose Baclofen Type Theory (BTT), a stripped-down version of CIC, together with a family of syntactic models of BTT that allows for a large range of effects in dependent type theory, amongst which exceptions, non-termination, non-determinism or writing operation. By syntactic models, we mean a model directly expressed in a type theory through a program transformation, as advocated in a previous paper [2].

The key feature of BTT lies in the fact it has a restricted version of dependent elimination to overcome the difficulty to marry effects and dependency. Essentially, the restriction appears as a side-condition on the return predicates of dependent pattern-matching, which must be *linear*, in the sense of Munch-Maccagnoni [7]. For instance, the elimination rule for booleans is of the following shape.

$$\frac{\Gamma \vdash M : \mathbb{B} \qquad \begin{array}{l}\Gamma \vdash N_1 : P\{b := \texttt{true}\} \\ \Gamma \vdash N_2 : P\{b := \texttt{false}\}\end{array} \qquad \Gamma, b : \mathbb{B} \vdash P : \square \qquad P \text{ linear in } b}{\Gamma \vdash \texttt{if}\ M\ \texttt{as}\ b\ \texttt{return}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2 : P\{b := M\}}$$

Linearity is a property of functions in an ambient call-by-name language. Intuitively, it captures the fact that a function is semantically call-by-value, or alternatively, in a more categorical parlance, that it is an algebra homomorphism. As showed by Levy in a recent paper [5], it is possible to provide syntactic underapproximations for linearity, so that the above side-condition can be understood as a *guard condition* similar to the one used for fixpoint productivity in practical CIC implementations. This guard condition is totally oblivious of the ambient effect and does not mention it at all. Furthermore, this restriction is a generalisation of the one we required for the call-by-name forcing translation [4], which was based on storage operators. It turns out that, at least in the non-recursive case, storage operators syntactically turn any predicate into a linear one.

The syntactic models are given by the *weaning translation* of BTT into CIC, using a variant of the traditional monadic translation. The need for this variant can be explained by analyzing the call-by-push-value (CBPV) decomposition of call-by-value and call-by-name reduction strategies. Indeed, the key observation is that the traditional monadic interpretation dating back to Moggi [6] is call-by-value whereas type theories such as CIC are fundamentally call-by-name because they feature an unrestricted conversion rule. Therefore, any effectful model of CIC ought to factor through a call-by-name decomposition in CBPV.

In fact, the weaning translation is somehow dual to the forcing translation through this decomposition, in the sense that they respectively trivialize the $\mathcal{U}$ and $\mathcal{F}$ functors which decompose the ambient monad `T` as an adjunction. Most notably, the weaning translation can be thought of as a variant of the Eilenberg-Moore construction on steroids, where types are translated as plain algebras (i.e. without coherence requirement), which can be easily expressed by the dependent sum

$$\boxed{\square}_i = \Sigma A : \square_i.\, \mathtt{T}\ A \to A.$$

But in CIC, universes satisfy a kind of self-enrichment expressed as $\square_i : \square_{i+1}$. Thus, to get a correct interpretation of universes, the monad needs to satisfy the additional requirement that the type of algebras needs to be itself an algebra of the monad. A monad satisfying this property is said to be *self-algebraic*. We then show how very common monads satisfy this property and thus give rise to effects that can be integrated to BTT. In particular, all free monads are also self-algebraic.

The exception monad is in particular self-algebraic, which allows us to adapt Friedman's A-translation to CIC. We recover the following theorem, which shows that Markov's rule is admissible in CIC, a fact which was, to the best of our knowledge, not known.

Theorem 1. *If* $\vdash_{\mathrm{CIC}} t : \neg\neg A$ *and* A *is a first-order type, then there exists* $\vdash_{\mathrm{CIC}} t^{\bullet} : A$.

As a matter of fact, in addition to weaning, BTT is also the source theory of our previous forcing translation, even though the two translations sit on two extreme points of CBPV decompositions. This leads us to postulate the following thesis.

BTT *models effectful type theories.*

As it is the case for other syntactic models [4, 2], it is possible to implement the weaning translation as a Coq plugin, thanks to the fact that it is a program translation preserving amongst other things conversion. The plugin is available at `https://github.com/CoqHott/coq-effects`, and allows to give the impression to the user that she lives in an impure theory while everything she writes is compiled on the fly to actual Coq terms.

References

[1] G. Barthe and T. Uustalu. Cps translating inductive and coinductive types. In *Proceedings of Partial Evaluation and Semantics-based Program Manipulation*, pages 131–142. ACM, 2002.

[2] S. Boulier, P.-M. Pédrot, and N. Tabareau. The next 700 syntactical models of type theory. In *Proceedings of Certified Programs and Proofs*, pages 182–194. ACM, 2017.

[3] H. Herbelin. On the degeneracy of sigma-types in presence of computational classical logic. In P. Urzyczyn, editor, *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2005.

[4] G. Jaber, G. Lewertowski, P.-M. Pédrot, M. Sozeau, and N. Tabareau. The definitional side of the forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 367–376, 2016.

[5] P. B. Levy. Contextual isomorphisms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 400–414, New York, NY, USA, 2017. ACM.

[6] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[7] G. Munch-Maccagnoni. Models of a Non-Associative Composition. In A. Muscholl, editor, *17th International Conference on Foundations of Software Science and Computation Structures*, volume 8412, pages 396–410, Grenoble, France, Apr. 2014. Springer.

A Small Basis for Homotopy Type Theory

Felix Rech[1] and Steven Schäfer[1]

Saarland University, Saarbrücken, Germany
{rech,schaefer}@ps.uni-saarland.de

Abbott et al. [1] show that nested strictly positive inductive and coinductive types can be constructed in an extensional Martin-Löf type theory with W-types. We report on a work-in-progress formalization and extension of these results to Homotopy Type Theory [4]. Our eventual goal is to simplify the construction of models for type theory, by reducing the problem of modeling all strictly positive types to the problem of modeling a small set of primitive types.

We work in a subset of the type theory from the HoTT book [4], which contains univalent universes closed under Π-, Σ-, and path-types together with a type of natural numbers and propositional resizing rules [5]. This is similar to the type theory underlying the UniMath project [6]. In particular, we do not assume the existence of W-types, but instead construct them using an impredicative encoding.

Constructing Non-Recursive Types Binary products and function types are special cases of Σ- and Π-types. The construction of binary coproducts and finite types is straightforward.

$$\mathbf{0} :\equiv 0 = 1 \qquad \mathbf{1} :\equiv \sum_{(n:\mathbb{N})} 0 = n \qquad \mathbf{2} :\equiv \sum_{(n:\mathbb{N})} n < 2 \qquad A + B :\equiv \sum_{(b:\mathbf{2})} \textbf{if } b \textbf{ then } A \textbf{ else } B.$$

We use $\mathbf{0}$ and $\mathbf{1}$ to define the *less than* relation on natural numbers by recursion and the conditional is syntactic sugar for the recursor on $\mathbf{2}$. All computation rules hold judgmentally.

Constructing M- and W-Types The constructions of Abbott et al. [1] are based on the notion of containers and container functors. A *container* $(S \rhd P)$ is a pair consisting of a type of shapes $S : \mathcal{U}$ and a family of position types $P : S \to \mathcal{U}$. The *extension* of a container is the functor $[\![S \rhd P]\!] :\equiv \lambda(X : \mathcal{U}).\ \sum(s : S), P(s) \to X$.

Inductive types can be constructed from the well-founded tree types, or W-types, which are the homotopy-initial algebras of containers [4]. Similarly, coinductive types can be constructed from M-Types, the homotopy-final coalgebras of containers. We build on the construction of M-types in HoTT by Ahrens et al. [2]. For a container $(S \rhd P)$, the construction of Ahrens et al. yields a type $M_{(s:S)}P(s)$ together with a corecursor corec and destructors label, arg

$$\begin{aligned} \mathsf{corec} &: \prod(C : \mathcal{U}), (C \to [\![S \rhd P]\!]\ C) \to C \to M_{(s:S)}P(s) \\ \mathsf{label} &: M_{(s:S)}P(s) \to S \\ \mathsf{arg} &: \prod(m : M_{(s:S)}P(s)), P(\mathsf{label}(m)) \to M_{(s:S)}P(s) \end{aligned}$$

with corresponding computation, uniqueness, and coherence laws. In general, we cannot expect the uniqueness or coherence laws to hold judgmentally. Using propositional resizing, we can however construct an equivalent type $M'_{(s:S)}P(s)$, the image of the corecursor, for which the computation rules hold judgmentally[1].

$$M'_{(s:S)}P(s) :\equiv \sum_{(m:M_{(s:S)}P(s))} \left\| \sum_{(C:U)} \sum_{(s:C\to[\![S\rhd P]\!]\ C)} \sum_{(c:C)} \mathsf{corec}(C, s, c) = m \right\|$$

[1]Where $\|A\| :\equiv \prod(P : \mathsf{Prop}), (A \to P) \to P$ is the propositional truncation of A.

We construct W-types as the subtype of well-founded trees of the corresponding M-type.

$$W_{(s:S)}P(s) :\equiv \sum_{(m:M_{(s:S)}P(s))} \mathsf{isWf}(m).$$

Writing M for $M_{(s:S)}P(s)$, we define the mere predicate isWf by

$$\mathsf{isWf}(m) :\equiv \prod_{(P:M\to\mathsf{Prop})} \left(\prod_{(m':M)} \left(\prod_{(b:B(\mathsf{label}(m')))} P(\mathsf{arg}(m',b)) \right) \to P(m') \right) \to P(m).$$

To prove initiality of $W_{(s:S)}P(s)$, we have to show that the type of $[\![S \rhd P]\!]$-algebra homomorphisms from $W_{(s:S)}P(s)$ is contractible. We first consider the type of local recursors $\mathsf{LHom}(m)$ for a fixed $m : M_{(s:S)}P(s)$, i.e., the type of $[\![S \rhd P]\!]$-algebra homomorphisms restricted to subtrees of m. For well-founded m, the type $\mathsf{LHom}(m)$ is contractible. We then show that the type of $[\![S \rhd P]\!]$-algebra homomorphisms from $W_{(s:S)}P(s)$ is a retract of the type $\prod(w : W_{(s:S)}P(s)), \mathsf{LHom}(\mathsf{pr}_1(w))$ and thus contractible.

Constructing strictly positive types. For the construction of strictly positive types we follow Abbott et al. [1]: First, we generalize containers to containers with parameters and then show that they are closed under constant functors, projections, binary products and coproducts, exponentiation as well as initial algebras and final coalgebras. Most constructions are without additional complications, except for the case of μ- and ν-containers, i.e., initial algebras and final coalgebras. For μ- and ν-containers we have to show additional coherence laws, since we do not have uniqueness of identity proofs.

Coq Development The current state of the formalization is available at `https://www.ps.uni-saarland.de/~rech/containers`. We are using the HoTT Library for Coq [3].

Future Work In the future we want to formalize indexed containers and use them to construct inductive and coinductive families. Furthermore, it might be interesting to consider extensions of containers to capture at least some higher-inductive types.

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[2] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in Homotopy Type Theory. *arXiv preprint arXiv:1504.02949*, 2015.

[3] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 164–172.

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

[5] Vladimir Voevodsky. Resizing rules. Talk at TYPES2011.

[6] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath*: Univalent Mathematics. Available at `https://github.com/UniMath`.

A Curry-Howard Approach to Church's Synthesis

Pierre Pradic[1,2] and Colin Riba[1]

[1] ENS de Lyon, Université de Lyon, LIP*
[2] University of Warsaw, Faculty of Mathematics, Informatics and Mechanics

Church's synthesis consists in the automatic extraction of stream transducers (or finite-state *synchronous* functions) from input-output specifications [4] (see also e.g. [8]). This problem has been solved by Büchi & Landweber [3] and Rabin [5] for specifications written in *Monadic Second-Order Logic* (MSO) on ω-words. MSO on ω-words is a decidable theory by Büchi's Theorem [2]. It subsumes non-trivial logics used in verification such as LTL.

Given an input-output specifications presented as an MSO-formula of the form $\forall\overline{X}\exists\overline{Y}\varphi(\overline{X};\overline{Y})$, where $\overline{X} = X_1, \ldots, X_p$ and $\overline{Y} = Y_1, \ldots, Y_q$ are second-order variables ranging over $\mathcal{P}(\mathbb{N}) \simeq \mathbf{2}^\omega$, the solutions of [3, 5] (see also e.g. [8]) in particular provide algorithms which decide if there exists a function $F : (\mathbf{2}^\omega)^p \to (\mathbf{2}^\omega)^q$ such that $\varphi(\overline{A}; F(\overline{A}))$ holds for all $\overline{A} \in (\mathbf{2}^\omega)^p$, and moreover such that F is implementable with a finite-state stream transducer. (We call this the *positive version* of Church's synthesis.)

Traditional approaches to synthesis (see e.g. [8]) suffer some drawbacks which make them unsuitable to practical implementations, namely, prohibitively high computational costs and limited compositionality. High complexities seems unavoidable when dealing with the full generality of MSO, or even LTL (see e.g. [1]). Having a fully compositional approach could help to mitigate this problem, in particular to help combining automatic methods with human intervention.

In this work, we propose a Curry-Howard approach to Church's synthesis. The Curry-Howard correspondence asserts that, given a suitable proof system, any proof therein can be interpreted as a program. Actually, *via* the Curry-Howard correspondence, the soundness of many type/proof systems is proved by means of *realizability*, which establishes how to read a formula from the logic as a specification for a program.

Our starting point is the fact that MSO on ω-words can be completely axiomatized [6]. From the classical axiomatization of MSO, we derive an intuitionistic system SMSO equipped with an extraction procedure which is sound and complete w.r.t. the positive version of Church's synthesis: proofs of existential statements can be translated to finite state synchronous realizers, and such proofs exist for all solvable instances of Church's synthesis. More precisely, we show the following (where $(-)^N$ is a negative translation).

*UMR 5668 CNRS ENS Lyon UCBL INRIA

Theorem 1. *Consider an* MSO*-formula* $\varphi(\overline{X};\overline{Y})$.

(i) *From a proof of* $\exists\overline{Y}\,\varphi^N(\overline{X};\overline{Y})$ *in* SMSO, *one can extract a finite-state synchronous realizer of* $\varphi(\overline{X};\overline{Y})$.

(ii) $\mathsf{SMSO} \vdash \exists\overline{Y}\varphi^N(\overline{X};\overline{Y})$ *if* $\varphi(\overline{X};\overline{Y})$ *admits a (finite-state) synchronous realizer.*

The key point in our approach is that on the one hand, finite-state realizers are constructively extracted from proofs in SMSO, while on the other hand, their correctness involves the full power of MSO. So in particular, our adaptation of the usual Adequacy Lemma of realizability does rely on the non-constructive proof of correctness of deterministic automata obtained by McNaughton's Theorem (see e.g. [7]), while these automata do not have to be concretely built during the extraction procedure.

In the course of proving Thm. 1, we provide an alternative way of extracting algorithmically realizers when they exist, thus completely revisiting the positive version of Church's synthesis problem. As it stands, we do not improve on the complexity of the existing algorithms solving the synthesis problem, but we do provide a compositional understanding of the problem by linking it to a notion of realizability.

In future work, we aim among other things at simplifying the complexity of our building blocks and at providing counter-strategies witnessing the impossibility of realizing a formula. Our outlook is that linear refinements of SMSO, coming from extensions of its realizability model, should be the suitable setting for these developments.

References

[1] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[2] J. R. Büchi. On a Decision Methond in Restricted Second-Order Arithmetic. In E. Nagel et al., editor, *Logic, Methodology and Philosophy of Science (Proc. 1960 Intern. Congr.)*, pages 1–11. Stanford Univ. Press, 1962.

[3] J. R. Büchi and L. H. Landweber. Solving Sequential Conditions by Finite-State Strategies. *Transation of the American Mathematical Society*, 138:367–378, 1969.

[4] A. Church. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic – Volume 1*, pages 3–50. Cornell Univ., Ithaca, N.Y., 1957.

[5] M. O. Rabin. Automata on infinite objects and Church's Problem. *Amer. Math. Soc.*, 1972.

[6] D. Siefkes. *Decidable Theories I : Büchi's Monadic Second Order Successor Arithmetic*, volume 120 of *LNM.* Springer, 1970.

[7] W. Thomas. Languages, Automata, and Logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume III, pages 389–455. Springer, 1997.

[8] W. Thomas. Solution of Church's Problem: A Tutorial. *New Perspectives on Games and Interaction*, 5:23, 2008.

Modelling Bitcoins in Agda

Anton Setzer

Dept. of Computer Science, Swansea University, Swansea, UK
`a.g.setzer@swan.ac.uk`

Abstract

We present a model of a block chain in Agda. We deal with Cryptographic operations and their correctness by postulating corresponding operations and their correctness. We determine correctness of blockchain transactions and show how to translate the blockchain back into a traditional ledger of a bank.

Since its introduction in November 2008, the market capitalisation of bitcoins has risen to over 18 Billion US-$. Other bitcoins such as Ethereum are following its lead. Cryptocurrencies have been proposed for introducing smart contracts. In its simplest form the buyer reserves money for the seller on the blockchain, and the seller only receives it once the seller has signed on time that she has received the goods. Bitcoins can be considered as the true cloud: whereas in normal cloud applications, data is stored on one server, and therefore everything relies on that service, Cryptocurrencies allow to store data on a peer-to-peer network. The block chain can then be used to certify which data is genuine, and determine the order and times when data was added.
In this project we use Agda as a modelling language for modelling the block chain. The goal is to obtain a deeper understanding of how the block chain operates and to prove correctness of certain aspects of the block chain. This project is the result of a series of third year and MSc projects supervised by the author at Swansea University. We follow the brown-bag talk by Warner [1], in which he shows how to obtain the blockchain starting from simple ledger.
In order to avoid having to introduce and verify cryptographic functions in Agda, we axiomatise those functions and their properties using Agda's postulates:

```
postulate Message : Set
postulate PublicKey : Set
postulate checkKey : (m : Message) (p : PublicKey) → Bool
postulate Names : Set
postulate messageToNat : (m : Message) → ℕ
postulate nameToPublicKey : (n : Names) → PublicKey
```

A message is here supposed to be a message with a signature and containing a value, namely the amount of bitcoins being represented in this message. checkKey checks whether a message has been signed by the private key corresponding to the public key of the name, and messageToNat determines the number contained in a message.
A bitcoin transaction consists of sequence of messages, and public keys, together with a proof that the messages have been signed by the private keys of the public keys:

```
data Input : Set where
  input : (message : Message) (publicKey : PublicKey)
          (cor : IsTrue (checkKey message publicKey)) → Input
```

similarly one can define outputs of a transaction. A transaction is now given by a list of inputs and a list of outputs:

```
data Transaction : Set where
  transaction : (input : List Input)(output : List Output) → Transaction
```

Time and amount of bitcoins are defined as natural numbers and the ledger is a function which assigns for every time and name the amount amount of bitcoins attributed to that person:

```
Time = ℕ
Amount = ℕ
Ledger = (t : Time)(n : Names) → Amount
```

We can now express the correctness of a transaction w.r.t. the state of the ledger before it is executed:

```
correctSingleTransaction : (oldLedger : Names → Amount)(trans : Transaction) → Set
correctSingleTransaction oldLedger (transaction inputlist outputlist)
  = IsTrue (checkKeysInInput inputlist)
    ∧ ((name : Names) →
                 IsTrue (oldLedger name ≥ sumOfInputs inputlist (nameToPublicKey name)))
    ∧ IsTrue (sumOfInputsTotal inputlist ≥ sumOfOutputsTotal outputlist )
    ∧ inputPublicKeysAreProper inputlist
    ∧ outputPublicKeysAreProper outputlist
```

We can compute now the ledger after one transaction:

```
updateSingleTransaction : (oldLedger : Names → Amount)(trans : Transaction)
                          (n : Names) → Amount
updateSingleTransaction oldLedger (transaction inputlist outputlist) n =
  oldLedger n - sumOfInputs inputlist (nameToPublicKey n)
             + sumOfOutputs outputlist (nameToPublicKey n)
```

and can compute from this the complete ledger from a sequence of transactions:

```
transactionsToLedger : (initialLedger : Names → Amount)(trans : Time → Transaction)
                       → Ledger
```

Modifications are needed in order to deal with mining and fees. We are currently working on extending this model to adding smart contracts. Modelling simple smart contracts is straightforward, the challenge is to introduce a language for more generalised smart contracts.

References

[1] B. Warner. Bitcoin: A technical introdution. Available from http://www.lothar.com/presentations/bitcoin-brownbag/, July 2011.

On Certain Group Structures in Type Theory

Sergei Soloviev

IRIT, University of Toulouse-3,
118, route de Narbonne, 31062, Toulouse, France,
soloviev@irit.fr

The "type theory" below means a system of typed λ-calculus. The notion of isomorphism of types is very abstract and may be defined in a uniform way for all systems under consideration. One needs only a sort of partial categorical structure: for all types A, B the class of terms that represent morphisms from A to B (usually one takes the terms $t : A \to B$); for each type A, a term id_A that represents identity (usually $\lambda x : A.x$); a composition $\circ$ (at least for terms $A \to B$ and $B \to A$); and an equivalence relation $\equiv$ on terms (usually the $\beta\eta$-eqivalence). The term t from A to B is an isomorphism iff there exists t^{-1} from B to A such that $t^{-1} \circ t \equiv id_A$ and $t \circ t^{-1} \equiv id_B$. The types A, B in this case are called isomorphic.

Isomorphisms of types in several largely known type theories such as simply typed λ-calculus $\lambda^1\beta\eta$, simply typed λ-calculus with surjective pairing and terminal object $\lambda^1\beta\eta\pi*$, second order polymorphic λ-calculus $\lambda^2\beta\eta$ and its extension with surjective pairing and terminal object $\lambda^2\beta\eta\pi*$ were studied in detail in [2]. Linear isomorphism of types was considered in [4]. Isomorphism in $\lambda^1\beta\eta\pi*$ extended with coproduct (disjunction) was considered in [3]. About isomorphism in dependent type systems, see for example [1, 5].

For a type A in each type theory where a notion of isomorphism is defined as above, one may define the groupoid $Gr(A)$ whose objects are the types $A' \sim A$ (all types isomorphic to A) and whose morphisms are the isomorphisms between such types, and the group $Aut(A)$ of automorphisms $A \to A$. (The elements of this group are λ-terms considered up to $\equiv$ and the group operation is the composition of λ-terms.)

It is, however, not the only group (groupoid) naturally associated with A. Let $FV(A) = \{a_1, ..., a_n\}$ be the set of free variables[1] of the type A and S_n the group of permutations of the set $\{1, ..., n\}$. For $\sigma \in S_n$ let $\sigma(A)$ denote the result of substitution $[a_{\sigma(1)}/a_1, ..., a_{\sigma(n)}/a_n]A$. We consider:

- the group of permutations $\Sigma(A) \subseteq S_n$ such that $\forall\sigma \in \Sigma(A).(\sigma(A) \sim A)$ (it is obvious that $\Sigma(A)$ is a group w.r.t. the composition of permutations);
- the groupoid $Gr_\Sigma(A) \subseteq Gr(A)$ whose objects are $A' \sim A$ such that $\exists\sigma \in \Sigma(A).(A' = \sigma(A))$ and morphisms are the same isomorphisms as in $Gr(A)$ (*i.e.*, it is a full subcategory of $Gr(A)$;
- other groups that are obtained by combination of isomorphisms and permutations that respect the isomorphism of types.

Example 1. Let $A = (a_1 \to a_2) \to (a_2 \to a_3) \to (a_3 \to a_1) \to a_4$. The permutation $(1, 2, 3)(4)$ (in cyclic notation) transforms A into $\sigma(A) = (a_2 \to a_3) \to (a_3 \to a_1) \to (a_1 \to a_2) \to a_4$, where $\sigma(A) \sim A$ because it is at the same time the result of permutation of premises. In fact, the group $\Sigma(A)$ is generated by σ. It is isomorphic to the cyclic group C_3. Other permutations, for example $\sigma' = (1, 2)(3)(4)$, do not belong to $\Sigma(A)$, that is $\sigma'(A)$ is not isomorphic to A. The group $Aut(A)$ is trivial. The groupoid $Gr(A)$ has 6 objects obtained from A by all

[1]The difference between free and bound variables in types is relevant for higher order type theories.

possible permutations of the premises $(a_1 \to a_2), (a_2 \to a_3)$ and $(a_3 \to a_1)$. As a consequence, there exist $A' \sim A$ such that for any $\rho \in S_n$ $A' \neq \rho(A)$.

We study the connections between these algebraic structures. In particular, we obtain the following results for the simply typed lambda-calculus $\lambda^1\beta\eta$ and for the second order system $\lambda^2\beta\eta$ (see [2]):

Theorem 1. For every finite group G there exists some type A in $\lambda^1\beta\eta$ such that the group $\Sigma(A)$ is isomorphic to G. It is possible to construct A in such a way, that at the same time $Aut(A) = \{id_A\}$.

Theorem 2. Let A be some type in $\lambda^1\beta\eta$ and $\overline{\forall}.A$ its universal closure (the type in the second order calculus $\lambda^2\beta\eta$). Then the group of automorphisms $Aut(\overline{\forall}.A)$ (in $\lambda^2\beta\eta$) is isomorphic to the cartesian product $Aut(A) \times \Sigma(A)$.

From these two theorems we derive the corollary that:

Corollary. For every finite group G there exists some type A in $\lambda^1\beta\eta$ such that the group $Aut(\overline{\forall}.A)$ (in $\lambda^2\beta\eta$) is isomorphic to G.

The proofs use a representation of *Cayley colored graphs* (known in group theory) by types in $\lambda^1\beta\eta$ and $\lambda^2\beta\eta$.

References

[1] Delahaye, D. (1999) Information Retrieval in a Coq Proof Library Using Type Isomorphisms. In *TYPES 1999, Lecture Notes in Computer Science*, **1956**, 131-147, Springer-Verlag.

[2] Di Cosmo, R. (1995) *Isomorphisms of types: from lambda-calculus to information retrieval and language design.* Birkhauser.

[3] Fiore, M. Di Cosmo, R., and Balat, V.(2006) Remarks on isomorphisms in typed lambda calculi with empty and sum types. *Annals of Pure and Applied Logic*, **141**(1),35-50.

[4] Soloviev, S.V.(1993) A complete axiom system for isomorphism of types in closed categories. In *A. Voronkov, ed., LPAR'93, Lecture Notes in Artificial Intelligence*, **698**, 360-371, Springer-Verlag.

[5] Soloviev, S. (2015) On Isomorphism of Dependent Products in a Typed Logical Framework. *Post-proceedings of TYPES 2014, LIPICS*, **39**, 275-288, Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Cumulative inductive types in Coq

Amin Timany[1], Matthieu Sozeau[2], and Bart Jacobs[1]

[1] imec-Distrinet, KU Leuven, Belgium, `firstname.lastname@cs.kuleuven.be`
[2] Inria Paris & IRIF, France, `matthieu.sozeau@inria.fr`

In order to avoid well-know paradoxes associated with self-referential definitions, higher-order dependent type theories stratify the theory using a countably infinite hierarchy of universes (also known as sorts), $\texttt{Set} = \texttt{Type}_0 : \texttt{Type}_1 : \cdots$. Such type systems are called cumulative if for any type T we have that $T : \texttt{Type}_i$ implies $T : \texttt{Type}_{i+1}$. The predicative calculus of inductive constructions (pCIC) [2, 3] at the basis of the Coq proof assistant, is one such system.

Earlier work [4] on universe-polymorphism in Coq allows constructions to be polymorphic in universe levels. The quintessential universe-polymorphic construction is the polymorphic definition of categories: $\texttt{Record}\ \texttt{Category}_{\texttt{i},\texttt{j}} := \{\texttt{Obj} : \texttt{Type}_{\texttt{i}};\ \texttt{Hom} : \texttt{Obj} \to \texttt{Obj} \to \texttt{Type}_{\texttt{j}};\ \cdots\}$.[1]

However, pCIC does not extend the subtyping relation (induced by cumulativity) to inductive types. As a result there is no subtyping relation between instances of a universe polymorphic inductive type. That is, for a category $\texttt{C}$, having both $\texttt{C} : \texttt{Category}_{\texttt{i},\texttt{j}}$ and $\texttt{C} : \texttt{Category}_{\texttt{i}',\texttt{j}'}$ is only possible if $\texttt{i} = \texttt{i}'$ and $\texttt{j} = \texttt{j}'$. In previous work Timany et al. [5] extend pCIC to pCuIC (predicative Calculus of Cumulative Inductive Constructions). This is essentially the system pCIC with a single subtyping rule added to it:[2]

C-Ind

$$\frac{\begin{array}{c} I \equiv (\mathsf{Ind}(X : \Pi\vec{x} : \vec{N}.\ s)\{\Pi\vec{x_1} : \vec{M_1}.\ X\ \vec{m_1}, \ldots, \Pi\vec{x_n} : \vec{M_n}.\ X\ \vec{m_n}\}) \\ I' \equiv (\mathsf{Ind}(X : \Pi\vec{x} : \vec{N'}.\ s')\{\Pi\vec{x_1} : \vec{M'_1}.\ X\ \vec{m'_1}, \ldots, \Pi\vec{x_n} : \vec{M'_n}.\ X\ \vec{m'_n}\}) \\ \forall i.\ N_i \preceq N'_i \qquad \forall i,j.\ (M_i)_j \preceq (M'_i)_j \qquad length(\vec{m}) = length(\vec{x}) \qquad \forall i.\ X\ \vec{m_i} \simeq X\ \vec{m'_i} \end{array}}{I\ \vec{m} \preceq I'\ \vec{m}}$$

The two terms I and I' are two inductive definitions (type constructors[3]) with indexes of types $\vec{N}$ and $\vec{N'}$ respectively. They are respectively in sorts (universes) s and s'. They each have n constructors, the i^{th} constructor being of type $\Pi\vec{x_i} : \vec{M_i}.\ X\ \vec{m_i}$ and $\Pi\vec{x_i} : \vec{M'_i}.\ X\ \vec{m'_i}$ for I and I' respectively. With this out of the way, the reading of the rule C-Ind is now straightforward. The type $I\ \vec{m}$ is a subtype of the type $I'\ \vec{m}$ if the corresponding parameters of corresponding constructors in I are sub types of those of I'. In other words, if the terms $\vec{v}$ can be applied to the i^{th} constructor of I to construct a term of type $I\ \vec{m}$ then the same terms $\vec{v}$ can be applied to the corresponding constructor of I' to construct a term of type $I'\ \vec{m}$. Using the rule C-Ind above (in the presence of universe polymorphism) we can derive $\texttt{Category}_{\texttt{i},\texttt{j}} \preceq \texttt{Category}_{\texttt{i}',\texttt{j}'}$ whenever $\texttt{i} \leq \texttt{i}'$ and $\texttt{j} \leq \texttt{j}'$.

The category theory library by Timany et al. [6] represents (relative) smallness and largeness of categories through universe levels. Smallness and largeness side-conditions for constructions are inferred by the kernel of Coq. In loc. cit. the authors prove a well-known theorem stating that any small and complete category is a preorder category. Coq infers that this theorem can apply to a category $\texttt{C} : \texttt{Category}_{\texttt{i},\texttt{j}}$ if $\texttt{j} \leq \texttt{i}$ and thus not to the category $\texttt{Types@\{i\}} : \texttt{Category}_{\texttt{i},\texttt{i+1}}$ of types at level $\texttt{i}$ (and functions between them) which is complete but not small. In a system with the rule C-Ind we have $\texttt{Types@\{i\}} : \texttt{Category}_{\texttt{k},\texttt{l}}$ for $\texttt{i} < \texttt{k}$, $\texttt{i} + \texttt{1} < \texttt{l}$ and $\texttt{l} \leq \texttt{k}$. However, subtyping would not allow for the proof of completeness of

[1]Records in Coq are syntactic sugar for an inductive type with a single constructor.
[2]The rule C-Ind is slightly changed here so that it applies to template polymorphism explained below.
[3]Not to be confused with constructors of inductive types

`Types@{i}` to be lifted as required. Intuitively, that would require the category to have limits of all functors from possibly larger categories.

Template Polymorphism Before the addition of full universe polymorphism to Coq, the system enjoyed a restricted form of polymorphism for inductive types, which was since coined template polymorphism. The idea was to give more precise types to applications of inductive types to their parameters, so that e.g. the inferred type of `list nat` is $\mathtt{Type}_0$ instead of $\mathtt{Type}_i$ for a global type level i.

Technically, consider an inductive type I of arity $\forall \vec{P}, \vec{A} \to s$ where $\vec{P}$ are the parameters and $\vec{A}$ the indices. When the type of the n-th parameter is $\mathtt{Type}_l$ for some level l and l occurs in the sort s (and nowhere else), the inductive is made parametric on l. When we infer the type of an application of I to parameters $\vec{p}$, we compute its type as $\forall \vec{A} \to s[l'/l]$ where $p_n : \mathtt{Type}_{l'}$, using the actual inferred types of the parameters.

This extension allows to naturally identify $\mathtt{list}(\mathtt{nat} : \mathtt{Set})$ and $\mathtt{list}(\mathtt{nat} : \mathtt{Type}_i)$ by convertibility, whereas with full universe polymorphism when comparing to $\mathtt{list@}\{\mathtt{Set}\}\ (\mathtt{nat} : \mathtt{Set})$ and $\mathtt{list@}\{i\}\ (\mathtt{nat} : \mathtt{Type}_i)$ with $\mathtt{Set} < i$ we would fail as equating i and $\mathtt{Set}$ is forbidden. With our new rule, this conversion will be validated as these two `list` instances become convertible. Indeed, convertibility on inductive applications will now be defined as cumulativity in both directions and in this case $\mathtt{list@}\{i\}$ cumulativity imposes no constraint on its universe variable. This change will allow a complete compatibility with template polymorphism.

Consistency and Strong Normalization The model constructed for pCIC by Lee et al. [3] is a set theoretic model that for inductive types considers the (fixpoints of the function generated by) constructors applied to all applicable terms. Therefore, the model readily includes all elements of the inductive types including those added by the rule C-Ind. Hence it is only natural to expect (and it is our conjecture that) the same model proves consistency of Coq when extended with the rule C-Ind. We are investigating using the abstract framework of B. Barras [1] to prove Strong Normalization with this extension.

Implementation The rule C-Ind above can be implemented in Coq very efficiently. The idea is that as soon as we define an inductive type, we compare two fresh instances of it (with two different sets of universe variables) to compute the set of constraints necessary for the subtyping relation to hold on different instances of that inductive type. Subsequent comparisons during type checking/inference will use these constraints. It is our plan to implement the rule C-Ind for the next release of Coq (Coq 8.7) and accordingly remove support for template polymorphism.

References

[1] Bruno Barras. *Semantical Investigation in Intuitionistic Set Theory and Type Theoris with Inductive Families.* PhD thesis, University Paris Diderot – Paris 7, 2012. Habilitation thesis.

[2] Coq Development Team. Coq reference manual, 2016. Available at https://coq.inria.fr/doc/.

[3] Gyesik Lee and Benjamin Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4), 2011.

[4] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Proceedings*, pages 499–514, 2014.

[5] Amin Timany and Bart Jacobs. First steps towards cumulative inductive types in CIC. In *Theoretical Aspects of Computing - ICTAC 2015, Proceedings*, pages 608–617, 2015.

[6] Amin Timany and Bart Jacobs. Category theory in coq 8.5. In *Conference on Formal Structures for Computation and Deduction, FSCD 2016, Proceedings*, pages 30:1–30:18, 2016.

Partiality and container monads

Tarmo Uustalu and Niccolò Veltri

Dept. of Software Science, Tallinn University of Technology, Tallinn, Estonia
{tarmo,niccolo}@cs.ioc.ee

In this work, we investigate monads of partiality in Martin-Löf type theory, following Moggi's general monad-based method for modelling effectful computations [8]. These monads are often called *lifting monads* and appear in category theory with different but related definitions. Here we revise some of them.

A *classifying monad* [7] on category $\mathbb{C}$ is a monad T with an operation $\overline{(-)}$, called *restriction*, sending any map $f : X \to T\,Y$ into $\overline{f} : X \to T\,X$, subject to the following conditions (where $\diamond$ is composition in the Kleisli category of T):

CM1 $f \diamond \overline{f} = f$,
CM2 $\overline{g} \diamond \overline{f} = \overline{f} \diamond \overline{g}$,
CM3 $\overline{g} \diamond \overline{f} = \overline{g \diamond \overline{f}}$,
CM4 $\overline{g} \diamond f = f \diamond \overline{g \diamond f}$,
CM5 $\overline{\eta_Y \circ f} = \eta_X$,
CM6 $\overline{\mathrm{id}_{TX}} = T\eta_X$.

CM1-4 stipulate that the Kleisli category of T is a restriction category; CM5 states that the left adjoint of the Kleisli adjunction is a restriction functor. CM6 is more technical. Together with CM5, the condition CM4 implies CM1.

A classifying monad is called *effective* [7], if, in addition, η is Cartesian, pullbacks along η_X exist and are preserved by T. Effective classifying monads are the same thing as *partial map classifiers* [7].

A strong monad on a category $\mathbb{C}$ with finite products, with strength $\psi_{X,Y} : X \times T\,Y \to T\,(X \times Y)$, is called *commutative*, if $\mu_{X\times Y} \circ T\,\psi^{\mathrm{rev}}_{X,Y} \circ \psi_{TX,Y} = \mu_{X,Y} \circ T\,\psi_{X,Y} \circ \psi^{\mathrm{rev}}_{X,TY}$. It is called an *equational lifting monad* [4], if, in addition, $\psi_{TX,X} \circ \Delta_{TX} = T\,((\eta_X \times \mathrm{id}_X) \circ \Delta_X)$. Every equational lifting monad is a classifying monad [7].

A *container* [1] is given by $S : \mathsf{Set}$ and $P : S \to \mathsf{Set}$. A container defines a set functor T by $T\,X = \Sigma s : S.\,P\,s \to X$. The functor T carries a monad structure iff it comes with certain extra structure [2], namely

- $\mathsf{e} : S$,
- $\bullet : \Pi s : S.\,(P\ s \to S) \to S$,
- $q_0 : \Pi s : S.\,\Pi v : P\,s \to S.\,P\,(s \bullet v) \to P\,s$,
- $q_1 : \Pi s : S.\,\Pi v : P\,s \to S.\,\Pi p : P\,(s \bullet v).\,P\,(v\,(q_0\,s\,v\,p))$

subject to a number of equational conditions.

We are interested in the question of when a container monad is a monad of any of the above types. We constrain the restriction to be $\overline{f} = T\,\mathsf{fst} \circ \psi_{X,Y} \circ (\mathrm{id}_X \times f) \circ \Delta_X$.

A container monad is a classifying monad iff it is a equational lifting monad iff the following three conditions hold:

A $s \bullet \lambda\_.\,s' = s' \bullet \lambda\_.\,s$,
B $P\,s \to s = \mathsf{e}$,
C $p = p'$ (each $P\,s$ is a proposition).

In the category of sets, all pullbacks exist and any container functor preserves all of them. The condition that η is Cartesian is equivalent to

D $P\,\mathsf{e}$.

So a container monad is an effective classifying monad iff, in addition to A+B+C, we have D.

Imposed simultaneously, the conditions A, B, C and D are very strong and constrain the monad very much. But when some of them are dropped, more examples arise.

The terminal monad ($T\,X =_{\mathrm{df}} 1$) is given by $S =_{\mathrm{df}} 1$, $P\,\star =_{\mathrm{df}} 0$. These data trivially satisfy A, B and C, but falsify D.

The maybe monad is given by $S =_{\mathrm{df}} \{\mathsf{ok}, \mathsf{err}\}$, $P\,\mathsf{ok} =_{\mathrm{df}} 1$, $P\,\mathsf{err} =_{\mathrm{df}} 0$, $\mathsf{e} =_{\mathrm{df}} \mathsf{ok}$, $\mathsf{ok} \bullet v =_{\mathrm{df}} v\,\star$, $\mathsf{err} \bullet \_ =_{\mathrm{df}} \mathsf{err}$. These data satisfy all of A, B, C and D.

The exception monad with two exceptions is given by $S =_{\mathrm{df}} \{\mathsf{ok}, \mathsf{err}_0, \mathsf{err}_1\}$, $P\,\mathsf{ok} =_{\mathrm{df}} 1$, $P\,\mathsf{err}_i =_{\mathrm{df}} 0$, $\mathsf{e} =_{\mathrm{df}} \mathsf{ok}$, $\mathsf{ok} \bullet v =_{\mathrm{df}} v\,\star$, $\mathsf{err}_i \bullet \_ =_{\mathrm{df}} \mathsf{err}_i$. These data satisfy B, C and D, but falsify A.

For Capretta's delay monad [5], we have $S =_{\mathrm{df}} \mathsf{co}\mathbb{N}$, $P\,n =_{\mathrm{df}} n{\downarrow}$. We can define $\mathsf{e} =_{\mathrm{df}} 0$, $0 \bullet v =_{\mathrm{df}} v\,0^{\downarrow}$, $\mathsf{suc}\,n \bullet v =_{\mathrm{df}} \mathsf{suc}\,(n \bullet \lambda(\mathsf{suc}^{\downarrow}\,p).\,v\,p)$. These data validate A, C and D, but not B. To modify this example to validate B too, we can quotient S by weak bisimilarity $\approx$ defined by $n \approx n' =_{\mathrm{df}} P\,n \leftrightarrow P\,n'$ resulting in a version of the Sierpinski space [6]. The definitions of P and e are easily adjusted to this change, but a weak choice principle is needed to adjust the definition of $\bullet$. A different version of the Sierpinski space can be defined from scratch as a higher inductive type (so it is by construction the free countably-complete join semilattice on the unit type) without recourse to the choice principle. The latter is equivalent to the Sierpinski space introduced by Altenkirch et al. using higher inductive-inductive types [3].

Acknowledgement We are thankful to Thorsten Altenkirch and Martín Escardó for discussions and valuable hints. This research was supported by Estonian Ministry of Education and Research institutional research grant IUT33-13.

References

[1] M. Abbott, T. Altenkirch, N. Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, v. 342, n. 1, pp. 3–27, 2005.

[2] D. Ahman, J. Chapman, T. Uustalu. When is a container a comonad? *Log. Methods in Comput. Sci.*, v. 10, n. 3, article 14, 2014.

[3] T. Altenkirch, N. A. Danielsson, N. Kraus. Partiality, revisited: The partiality monad as a quotient inductive-inductive type. In J. Esparza, A. Murawski, eds., *Proc. of 20th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2017*, v. 10203 of *Lect. Notes in Comput. Sci.*, pp. 534–549. Springer, 2017.

[4] A. Bucalo, C. Führmann, A. Simpson. An equational notion of lifting monad. *Theor. Comput. Sci.*, 294(1–2), 31–60, 2003.

[5] V. Capretta. General recursion via coinductive types. *Log. Meth. in Comput. Sci.*, v. 1, n. 2, article 1, 2005.

[6] J. Chapman, T. Uustalu, N. Veltri. Quotienting the delay monad by weak bisimilarity. In M. Leucker, C. Rueda, F. D. Valencia, eds., *Proc. of 12th Int. Coll. on Theoretical Aspects of Computing, ICTAC 2015*, v. 9399 of *Lect. Notes in Comput. Sci.*, pp. 110–125. Springer, 2015.

[7] J. R. B Cockett, S. Lack. Restriction categories II: partial map classification. *Theor. Comput. Sci.*, 294(1–2), 61–102, 2003.

[8] E. Moggi. Notions of computation and monads. *Inf. and Comput.*, 93(1), 55–92, 1991.

Expressive and Strongly Type-Safe Code Generation

Thomas Winant, Jesper Cockx, and Dominique Devriese

imec-DistriNet, KU Leuven
`firstname.lastname@cs.kuleuven.be`

Meta-programs are useful to avoid writing boilerplate code, for polytypic programming, etc. However, when a meta-program passes the type checker, it does not necessarily mean that the programs it generates will be free of type errors, only that generating object programs will proceed without type errors. For instance, this well-typed Template Haskell [5] meta-program generates the ill-typed object program *not* `'X'`.

```
notX :: Q Exp
notX = [| not 'X' |]
```

Fortunately, Template Haskell will type-check the generated program after generation, and detect the type error. We call such meta-programming systems *weakly type-safe.* Even though weakly type-safe meta-programming suffices for guaranteeing that the resulting program is type-safe, it has important downsides. Type errors in the generated code are presented to the application developer, who may simply be a user of the meta-program. If the meta-program is part of a library, the developer has little recourse other than contacting the meta-program authors about the bug in their code. Moreover, composing weakly type-safe meta-programs can be brittle because the type of generated programs is not specified in a machine-checked way.

We are interested in what we call *strongly type-safe* meta-programming, which offers a stronger guarantee: when a meta-program is strongly type-safe, all generated programs are guaranteed to be type-safe too. As such, bugs in meta-programs are detected as early as possible. In fact, all arguments in favour of static typing can be made.

Existing strongly type-safe meta-programming systems like MetaML [6], Typed Template Haskell [3], and Scala's reflection API [1] offer this guarantee by providing *typed quotations*, i.e. quotations that are type-checked at meta-program compile-time. For example, when the faulty meta-program is rewritten using a typed quotation, the bug is detected at meta-program compile time.

```
notX :: Q (TExp Bool)
notX = [|| not 'X' ||]
```

Unfortunately, to offer this guarantee, these and other systems compromise on expressiveness [6, 2, 4]. In particular, while typed object expressions can be constructed using typed quotations, their *types* and *typing contexts* cannot. These are severe restrictions that make it impossible to develop strongly type-safe variants of many common weakly type-safe meta-programs.

A real world example that suffers from this restriction is the generation of *lenses* [7] for a record data-type. Such a meta-program is not expressible using state-of-the-art strongly type-safe meta-programming systems. To illustrate this restriction, consider a simplified sketch of what the Typed Template Haskell variant of *deriveLenses* would look like:

```
deriveLenses adt = map (λfield → deriveLens adt field) (fields adt)
deriveLens :: ADT → Field → Q (TExp (Lens ? ?))
deriveLens adt field = ...
```

The question marks should be replaced with the type of the record data type and the type of the field. Clearly, the type of the generated lens *depends* on the types of the record data type's fields, which are only available at the value level in the meta-program. If Haskell had *dependent types*, one could write this signature:

$$deriveLens :: (adt :: ADT) \to (f :: Field\ adt) \to Q\ (TExp\ (LensType\ adt\ f))$$

Where *LensType* is a type-level function that calculates the type of the lens. Such a syntactic construction of the type of an object program is fundamentally impossible in Typed Template Haskell and other MetaML-like systems. Deeper inside the implementation of *deriveLens*, it gets worse as the types of generated expressions depend in more complex ways on the values *adt* and *f*, and they are also constructed in *contexts* that depend on them. The underlying reason for this limited expressiveness is that the meta-level type system of these systems is not powerful enough to express the naturally dependent types of many strongly type-safe meta-programs.

We propose a new design that delivers strong type-safety without compromising on expressiveness. Our first key design choice is to represent object programs by an inductive type family in an off-the-shelf dependently-typed language (Agda). This type family is indexed by the type of the program, and its type and variable contexts. Each of its constructors encodes one language construct, including its corresponding typing rule. Using this encoding, meta-programs construct object programs that are correct by construction. This approach is standard in dependently typed languages, yet it isn't commonly used by existing meta-programming systems.

Our second key design choice is to use a small explicitly-typed core language as the object language (in our case GHC Core), instead of the full surface language (which would be Haskell). This choice is based on the observation that surface languages are designed for programmers, not meta-programs. Their complex syntax, typing rules, type inference, and tendency to change make them ill-suited as object languages. In contrast, a core language such as GHC Core is designed to be used by the compiler. As a consequence, it is typically well-studied, small, explicitly typed, relatively stable, and has a full formal description, while remaining relatively close to the surface language. Using a core language instead of the full surface language is not an academic simplification, but a feature of our approach: it is a central design choice that we believe is essential to make our approach realistic to implement and use.

Our approach to strongly type-safe metaprogramming is based on existing technology (an off-the-shelf dependently-typed language and a standard encoding of the object language) but applies it in a new way. We have implemented is as a proof of concept for Haskell. Using our implementation, we developed strongly type-safe variants of existing real-world meta-programs: deriving lenses and deriving the `Eq` type class. In a fair comparison with the original meta-programs, our meta-programs count roughly the same number of SLOC. This shows that our approach is practical as well as simple, expressive, and strongly type-safe.

References

[1] Eugene Burmako. *Unification of Compile-Time and Runtime Metaprogramming in Scala.* PhD thesis, EPFL, 2017.

[2] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *ICFP*. ACM, 2003.

[3] Geoffrey Mainland. Type-safe runtime code generation with (Typed) Template Haskell. https://www.cs.drexel.edu/~mainland/2013/05/31/type-safe-runtime-code-generation-with-typed-template-haskell/.

[4] Geoffrey Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In *ICFP*. ACM, 2012.

[5] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12), December 2002.

[6] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12), December 1997.

[7] Twan van Laarhoven. CPS based functional references. http://twanvl.nl/blog/haskell/cps-functional-references, 2009.

Parallelization of Software Verification Tool LAV

Branislava Živković and Milena Vujošević Janičić

Faculty of Mathematics, University of Belgrade, Serbia
{branislavaz | milena} @ matf.bg.ac.rs

Automated software verification, based on model checking is powerful, but typically very time and memory consuming. The two most demanding steps are (i) constructing a formula representing program executions and correctness conditions of these executions, and (ii) checking validity of generated formula by an automated theorem prover. One way to speed this process up is to use parallelization techniques which take advantage of the underlying hardware architecture: computer systems nowadays have several cores. The dominating approach is to parallelize the algorithm used by theorem prover, for example paralellization of SMT solving [2] or parallelization of state exploration in model checking [8, 3]. However, for complex programs, already constructing the formula representing correctness conditions may be very demanding.

We present our ongoing work on parallelization of software verification tool LAV [6]. LAV is an open-source tool (implemented in C++ and publicly available[1]) for statically verifying program assertions and detecting bugs such as buffer overflows, pointer errors and division by zero. LAV uses the popular LLVM infrastructure and combines bounded model checking, symbolic execution, and SAT encoding of program's control-flow to construct correctness conditions in form of first order logical formulae [5], which are then checked by an external SMT solver. LAV was successfully applied on different benchmarks [6] and also used in automated evaluation of students' programs [7]. More about overall LAV architecture can be found in literature [6, 5]. LAV can generate a separate correctness condition for each potentially unsafe command or assertion. This makes LAV suitable for experimenting with different kinds of parallelizations. So far, we implemented parallel verification of different functions and parallel checking of correctness conditions within one block of code.

For testing scalability of these parallelizations, we constructed a custom corpus consisting of five different sets with 2020 programs in total (in programming language C).[2] First four sets, named as S_1, S_2, S_3, and S_4, contain 500 of programs each, and neither one of them contains bugs (so it is necessary to search over all feasible states). In each set, the programs have the increasing number of functions and the increasing number of commands inside each function, yielding levels of complexity denoted from 1 to 10. In each function there is one command that should be checked for a division by zero bug. In sets S_1 and S_2 there is one level of function calls (the function main calls all these functions), while in sets S_3 and S_4 each function calls another function (of a similar structure and complexity). In sets S_1 and S_3 the division is not depending on calculations performed in previous commands, while in sets S_2 and S_4 this division depends on calculations performed in previous commands (making it more difficult to reason about). The fifth set of programs contains 20 programs which all contain a division by zero bug and differ by the number of commands.

We ran the experiments on a cluster computer with 48 cores and 94GiB of memory, running Ubuntu 16.04. We used the LAV tool, and also the model checker CBMC [4] version 5.5. The time out was set to 400 seconds. The experimental results for the described four sets are shown in Figure 1. The results show that CBMC verification times-out in the third and the fourth set when the programs reach the level of complexity 7. In these cases, CBMC timed-out already at the level of constructing a formula (it did not get to calling the solver). LAV

[1] http://argo.matf.bg.ac.rs/?content=lav
[2] The corpus and the scripts used for the experiments are available from the LAV page.

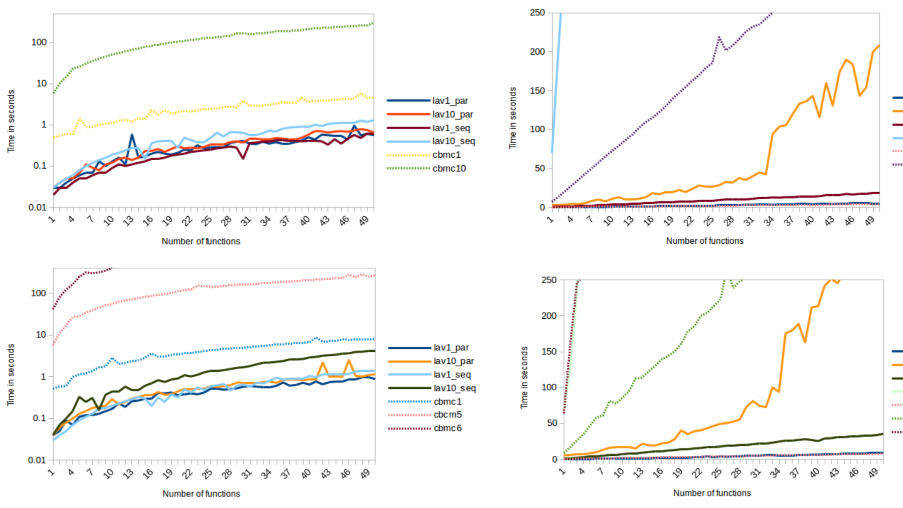


Figure 1: Experimental results comparing CBMC and LAV on four different sets of programs (set S_1: left-top, set S_2: right-top, set S_3: left-bottom, set S_4: right-bottom).

without parallelization times out at set S_2 at the level 7 and at the set S_4 at the level 6. On the other hand, LAV with parallelization successfully handled all the programs. In the fifth corpus, CBMC timed-out at the program with 28 commands, while parallelization within LAV kept the time below one second even for programs that are double in size. Namely, the formula was divided into different threads and in one of these threads solver easily finds the bug.

The presented experimental evaluation shows that parallelization may scale well in cases where classical model checking times out, so there is a lot of room for making improvements in this context. Our future work is to broaden the set of usable parallelizations, and to make experiments on wider corpus, for example to evaluate it on SV-COMP benchmark [1]. We are also planning to formally describe and analyze the parallelization techniques that we used.

Acknowledgements This work was partially supported by the Serbian Ministry of Science grant 174021 and by COST action CA15123.

References

[1] Competition on Software Verification, 2017. on-line at: `https://sv-comp.sosy-lab.org/2017/`.

[2] M. Banković. Parallelizing simplex within smt solvers. *Artificial Intellig. Review*, pages 1–30, 2016.

[3] J. Barnat, L. Brim, and P. Ročkai. *Scalable Multi-core LTL Model-Checking*, pages 187–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[4] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176. Springer, 2004.

[5] M. Vujošević Janičić. Modelling Program Behaviour within Software Verification Tool LAV. In *Type Theory Based Tools*, 2017.

[6] M. Vujošević Janičić and V. Kuncak. Development and Evaluation of LAV: An SMT-Based Error Finding Platform. In *VSTTE*, LNCS, 2012.

[7] M. Vujošević Janičić, M. Nikolić, D. Tošić, and V. Kuncak. Software verification and graph similarity for automated evaluation of students assignments. *Information and Softw. Technology*, 55(6), 2013.

[8] A. W. Laarman. *Scalable multi-core model checking*. PhD thesis, Centre for Telematics and Information Technology, University of Twente, 2014.

Author Index