TYPES 2016

Types for Proofs and Programs

22nd Meeting

———————

Novi Sad, Serbia

23 – 26 May, 2016

———————

Book of Abstracts

Preface

This volume contains the abstracts of the talks presented at the 22nd International Conference on Types for Proofs and Programs, TYPES 2016 held in Novi Sad, Serbia, 23-26 May 2016.

The TYPES meetings are a forum to present new and on-going work in all aspects of type theory and its applications, especially in formalized 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).

The TYPES areas of interest include, but are not limited to: foundations of type theory and constructive mathematics; homotopy type theory; 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 conference is internationally open and has an informal character. The contributions are based on work in progress and already presented or published work on newly published papers, work submitted for publication. The selection of contributed talks is based on short abstracts that present new results, work in progress or newly published, submitted or communicated work.

The programme of TYPES 2016 comprises three invited lectures by Simon Gay, Dale Miller and Simona Ronchi Della Rocca. The contributed part of the programme consists of 46 talks. Each submission was reviewed by three programme committee members, who were assisted in their work by external reviewers.

TYPES 2016 formal post-proceedings will appear in Dagstuhl's Leibniz International Proceedings in Informatics (LIPIcs) series, similarly to the editions of the conference since 2011. It will be prepared after the conference and the submitted full papers presenting unpublished work will undergo a complete review process.

University of Novi Sad, Faculty of Technical Sciences and Mathematical Institute SANU have been the host and the organizers of TYPES 2016.

Many people helped to make TYPES 2016 a success. Thanks go to the authors of submissions whose contributions shaped the scientific content of the meetings. The effort of the programme committee members who donated their time to TYPES 2016 by careful and active reviewing is gratefully acknowledged. Herman Geuvers, Chair of the TYPES Steering Committee and Chair of the EUTypes COST action, put a lot of effort to bring together the meeting and the action. EasyChair facilitated the reviewing process and the generation of this

book of abstracts. The local organizing team did a good job in carrying out the organization.

TYPES 2016 is supported by the COST Action CA15123 EUTypes and RT-RK Company.

April 21, 2016
Novi Sad

Silvia Ghilezan
Jelena Ivetić

Table of Contents

Program Committee

Thorsten Altenkirch	University of Nottingham
Zena Ariola	University of Oregon
Andrej Bauer	University of Ljubljana
Marc Bezem	University of Bergen
Malgorzata Biernacka	University of Wroclaw
Edwin Brady	University of St Andrews
Thierry Coquand	University of Gothenburg
Jose Espirito Santo	University of Minho
Ken-Etsu Fujita	Gunma University
Silvia Ghilezan	University of Novi Sad, Faculty of Technical Sciences
Hugo Herbelin	INRIA Paris-Rocquencourt
Jelena Ivetic	University of Novi Sad
Marina Lenisa	University of Udine
Elaine Pimentel	Federal University of Rio Grande do Norte
Andrew Polonsky	VU University Amsterdam
Jakob Rehof	Technical University of Dortmund
Claudio Sacerdoti Coen	University of Bologna
Carsten Schürmann	IT University of Copenhagen
Wouter Swierstra	Utrecht University
Nicolas Tabareau	INRIA
Tarmo Uustalu	Tallinn University of Technology

Additional Reviewers

B

Benzmüller, Christoph
D

Di Gianantonio, Pietro
F

Ferreira, Fernando
H

Honsell, Furio
K

Kurata, Toshihiko
M

Maksimovic, Petar
Maurer, Luke
Miculan, Marino
Møgelberg, Rasmus Ejlers
N

Nigam, Vivek
O

Olarte, Carlos
R

Reis, Giselle
S

Scagnetto, Ivan
Y

Yokouchi, Hirofumi

Session Types: Achievements and Challenges

Simon J. Gay

School of Computing Science, University of Glasgow, UK
Simon.Gay@glasgow.ac.uk

Abstract

Session types are type-theoretic specifications of communication protocols, introduced by Kohei Honda and collaborators in the mid-1990s. They define the type and sequence of messages exchanged via a communication medium, and allow type-checking techniques to be used to verify protocol implementations. Whereas data types codify the static structure of information in a computer program, session types codify the dynamic structure of communication in a software system. The classic slogan *"algorithms + data structures = programs"* can be generalised to *"programs + communication structures = systems"*, and the full range of type-checking technology can be generalised too.

In the simplest form, a session type specifies a straightforward sequence of messages. The type $!\mathsf{int}.?\mathsf{bool}.\mathsf{end}$ describes how to run a protocol on an endpoint of a communication channel: first send (!) an integer, then receive (?) a boolean, then terminate. The other endpoint has the dual type $?\mathsf{int}.!\mathsf{bool}.\mathsf{end}$. More complex protocols include choice and repetition. For example, the recursive type S defined by $S = \&\langle \mathsf{start} : ?\mathsf{int}.!\mathsf{bool}.S, \mathsf{stop} : \mathsf{end}\rangle$ describes a protocol that offers a choice between start and stop, each with its own continuation protocol. The basic idea for protocol verification is to match the structure of a session type with the use of communication operations in a program.

The twenty years since the introduction of session types have seen a dramatic growth in research activity. There is now a substantial community, and most programming-language-related conferences regularly include papers on session types. Several themes of research can be identified:

- Generalisation of session types from two-party to multi-party sessions.
- Transfer of session types from pi-calculus to a range of programming language paradigms.
- Logical foundations of session types via a Curry-Howard correspondence with linear logic.
- Connections between session types and automata theory.
- Development of programming language implementations and session-type-based tools.
- Broadening the original focus on static type-checking to include dynamic monitoring.
- Incorporation of time and error-handling.
- Connections with more general type-theoretic concepts such as dependent types, gradual types and typestate.

The lecture will introduce session types, survey the main themes and achievements of the field, and suggest directions for future work that are likely to be of interest to researchers from the wider area of type theory.

Mechanized metatheory revisited (abstract)

Dale Miller

Inria & LIX/École polytechnique
dale.miller@inria.fr

Over a decade ago, the POPLmark challenge [2] suggested that the theorem proving community had tools that were close to being usable by programming language researchers to formally prove properties of their designs and implementations. The authors of the POPLmark challenge looked at existing practices and systems and urged the developers of proof assistants to make improvements to existing systems.

> Our conclusion from these experiments is that the relevant technology has developed almost to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research—mechanized metatheory for the masses. [2]

In fact, a number of research teams have used proof assistants to formally proved significant properties of entire programming languages. Such properties include type preservation, determinancy of evaluation, and the correctness of an OS microkernel and of various compilers: see, for example, [9, 10, 11, 15].

As noted in [2], the poor support for binders in syntax was one problem that held back proof assistants from achieving even more widespread use by programming language researchers and practitioners. In recent years, a number of extensions to programming languages and to proof assistants have been developed for treating bindings. These go by names such as locally nameless [4, 18], nominal reasoning [1, 5, 17, 19], and parametric higher-order abstract syntax [6]. Some of these approaches involve extending underlying programming language implementations while the others do not extend the proof assistant or programming language but provide packages, libraries, and/or abstract datatypes that attempt to hide and orchestrate various issues surrounding the syntax of bindings. In the end, nothing canonical seems to have appeared since the POPLmark challenge was made: we are left with a simple grid that rates different approaches on various attributes [16].

Clearly, mature and extensible proof assistants, such as, say, Coq, HOL, and Isabelle/HOL can be extended to deal with syntactic challenges (such as bindings in syntax) that they were not originally designed to handle. At the same time, it seems plausible and desirable to pursue approaches to the problem of bindings in syntax and metatheory more generally.

In this talk, I will argue that bindings are such an intimate aspect of the structure of expressions that they should be accounted for directly in the underlying programming language support for proof assistants. High-level and semantically elegant programming language support can be found in rather old and familiar concepts. In particular, Church's Simple Theory of Types [7] has long ago provided answers to how bindings interact with logical connectives and quantifiers. Similarly, the proof search interpretation [13] of Gentzen's proof theory [8] provides a rich model of computation that supports bindings. I outline several principles for dealing computationally with bindings that follow from their treatments in quantificational logic and sequent calculus. One of the most central principles about bindings is that bound variables never become free: instead bindings can move from term-level bindings (λ-abstractions) to formula-level bindings (quantifiers) to proof-level bindings (eigenvariables and nominal constants) [12, 14] I will also describe some implementations [3, 12] of these principles that have helped to validate their effectiveness as computational principles.

Acknowledgments. This work has been funded by the ERC Advanced Grant ProofCert.

References

[1] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. In *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP)*, pages 69–77, Seattle, WA, USA, August 2006.

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer, 2005.

[3] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.

[4] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, May 2011.

[5] James Cheney and Christian Urban. Nominal logic programming. *ACM Trans. Program. Lang. Syst.*, 30(5):1–47, 2008.

[6] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008.

[7] Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

[8] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935.

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR)*, pages 207–220, Big Sky, MT, October 2009. ACM SIGOPS.

[10] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[11] Donald MacKenzie. *Mechanizing Proof*. MIT Press, 2001.

[12] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

[13] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[14] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.

[15] J. Strother Moore. A mechanically verified language implementation. *J. of Automated Reasoning*, 5(4):461–492, 1989.

[16] The POPLmark Challenge webpage. `http://www.seas.upenn.edu/~plclub/poplmark/`, 2015.

[17] François Pottier. An overview of Cαml. In *ACM Workshop on ML*, ENTCS, pages 27–51, September 2005.

[18] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(01):71–122, 2010.

[19] Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

Intersection Types for denotational semantics

Simona Ronchi Della Rocca

Dipartimento di Informatica, Università di Torino, Italy

Intersection types can supply a tool for reasoning in a finitary way about the denotation of terms in models of lambda calculus. I will show how intersection type assignment systems can be tailored in such a way of describing denotational interpretation of terms in three very general classes of λ-models, namely Scott models, based on domains of continuous functions, Girard models, based on the domains of stable functions, and relational models, based on the category of sets and relations, recently defined by Ehrhard, Bucciarelli and Manzonetto, starting from an idea of Selinger. Each one of the three classes is described by a parametric type assignment system, when a particular model is grasped by a suitable instance of the parameter. The three parametric type assignment systems share a similar structure, but they differ not only in the parametric rules, but also in the properties of the intersection and in the structural rules. Namely, intersection needs to be idempotent to describe functional semantics, but not idempotent to describe relational semantics, weakening is necessary for the continuous semantics, while it is unsound in both the stable and relational semantics. Despite the differences between the various approaches, in each case the problem asking if two closed terms are equal in a model reduces to that one of asking if they can be assigned the same set of types in a given type assignment system. There are finitary techniques for solving the last problem: so intersection types supply a tool for proving semantical properties of terms, which can be applied in a uniform way to different kinds of models.

On the Decidability of Conversion in Type Theory

Andreas Abel, Thierry Coquand, and Bassel Mannaa

Department of Computer Science and Engineering, University of Gothenburg, Gothenburg, Sweden
{andreas.abel,thierry.coquand,bassel.mannaa}@cse.gu.se

We present a proof of decidability of conversion in dependent type theory with natural numbers N, a universe U à la Russell, and functional extensionality. Our approach is based on one by Abel and Scherer [1] with the difference that we use *typed* weak head reduction and thus get subject reduction for free. First we define one logical relation and prove it sound and complete w.r.t. the type system. We then obtain canonicity, injectivity of Π, and unicity of types for neutral terms. We then define algorithmic equality and show it is decidable. We define a second logical relation and show it is sound and complete w.r.t. the type system and algorithmic equality. We thus conclude that conversion is decidable in the type system.

Typed weak head reduction $\Gamma \vdash t \to t' : A$, which is a subrelation of conversion $\Gamma \vdash t = t' : A$, is given by:

$$\frac{\Gamma \vdash t \to t' : \Pi(x{:}A)B \qquad \Gamma \vdash a : A}{\Gamma \vdash t\,a \to t'a : B[a]} \qquad \frac{\Gamma, x{:}A \vdash t : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.t)\,a \to t[a/x] : B[a]}$$

$$\frac{\Gamma \vdash t \to u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t \to u : B} \qquad \frac{\Gamma \vdash A \to B : \mathsf{U}}{\Gamma \vdash A \to B}$$

We also define the usual reflexive-transitive closures $\to^*$ of these relations.

We define a logical relation by mutual induction-recursion. Inductively we define $\Gamma \Vdash A$ by introduction rules. By recursion on the proofs of $\Gamma \Vdash A$ we define $\Gamma \Vdash a : A$, $\Gamma \Vdash a = b : A$ and $\Gamma \Vdash A = B$.

Definition 1. $\textsc{f-N}\ \dfrac{\Gamma \vdash A \to^* \mathsf{N}}{\Gamma \Vdash A} \qquad \textsc{f-Neu}\ \dfrac{\Gamma \vdash A \to^* K \qquad K \text{ neutral}}{\Gamma \Vdash A} \qquad \textsc{f-U}\ \dfrac{\Gamma \vdash A \to^* \mathsf{U}}{\Gamma \Vdash A}$

$$\textsc{f-}\Pi\ \frac{\Gamma \vdash A \to^* \Pi(x{:}F)G \quad \Gamma \Vdash F \quad (\forall a, \forall \Delta \le \Gamma)(\Delta \Vdash a : F \Rightarrow \Delta \Vdash G[a]) \quad (\forall a, b, \forall \Delta \le \Gamma)(\Delta \Vdash a = b : F \Rightarrow \Delta \Vdash G[a] = G[b])}{\Gamma \Vdash A}$$

- If $\Gamma \Vdash A$ by f-Neu ($A \to^* K$ with K neutral) then
  - $\Gamma \Vdash A = B$ if $\Gamma \vdash B \to^* L$ with L neutral and $\Gamma \vdash K = L$.
  - $\Gamma \Vdash t : A$ if $\Gamma \vdash t \to^* l : A$ with l neutral.
  - $\Gamma \Vdash t = u : A$ if $\Gamma \vdash t \to^* l : A$ and $\Gamma \vdash u \to^* k : A$ with l and k neutrals and $\Gamma \vdash l = k : A$.
- If $\Gamma \Vdash A$ by f-N then
  - $\Gamma \Vdash A = B$ if $\Gamma \vdash B \to^* \mathsf{N}$.
  - $\Gamma \Vdash t : A$ if one of (i.) $\Gamma \vdash t \to^* 0 : A$ (ii.) $\Gamma \vdash t \to^* \mathsf{S}\,u : A$ and $\Gamma \Vdash u : A$ (iii.) $\Gamma \vdash t \to^* k : A$ with k neutral.
  - $\Gamma \Vdash t = u : A$ if one of (i.) $\Gamma \vdash t \to^* 0 : A$ and $\Gamma \vdash u \to^* 0 : A$ (ii.) $\Gamma \vdash t \to^* \mathsf{S}\,t' : A$, $\Gamma \vdash u \to^* \mathsf{S}\,u' : A$ and $\Gamma \Vdash t' = u' : A$ (iii.) $\Gamma \vdash t \to^* k : A$ and $\Gamma \vdash u \to^* l : A$ with l and k neutral and $\Gamma \vdash k = l : A$.

- If $\Gamma \Vdash A$ by F-Π with $\Gamma \vdash A \to^* \Pi(x{:}F)G$ then
  - $\Gamma \Vdash A = B$ if $\Gamma \Vdash B$ with $\Gamma \vdash B \to^* \Pi(x{:}H)E$ and $\Gamma \Vdash F = H$ and $\Delta \Vdash G[a] = E[a]$ whenever $\Delta \Vdash a : F$ for any a and $\Delta \leq \Gamma$.
  - $\Gamma \Vdash f : A$ if $\Gamma \vdash f : A$ and $\Delta \vdash f\,a : G[a]$ whenever $\Delta \Vdash a : F$, and $\Delta \vdash f\,a = f\,b : G[a]$ whenever $\Delta \Vdash a = b : F$ for any a, b and $\Delta \leq \Gamma$.
  - $\Gamma \Vdash f = g : A$ if $\Gamma \Vdash f : A$ and $\Gamma \Vdash g : A$ and $\Delta \Vdash f\,a = g\,a : G[a]$ whenever $\Delta \Vdash a : F$ for any a and $\Delta \leq \Gamma$.

Formally, $\Gamma \Vdash A$ is the type of a derivation. E.g., the rule F-N should be written as $\dfrac{\mathcal{R} :: \Gamma \vdash A \to^* \mathsf{N}}{\mathrm{tf}_{\Gamma,\mathrm{A}}(\mathcal{R}) :: \Gamma \Vdash A}$ where $\mathcal{R}$ is a proof of $\Gamma \vdash A \to^* \mathsf{N}$. The forcing $\Gamma \Vdash t : A$ is then given by a map $\mathrm{trm} : (\Gamma \in \mathbb{C},\ A \in \mathbb{T},\ \mathcal{F} \in (\Gamma \Vdash A)) \to \mathcal{P}(\mathbb{T}))$ defined by recursion on $\mathcal{F}$, where $\mathbb{C}$ and $\mathbb{T}$ are the sets of contexts and terms and $\mathcal{P}$ denotes the powerset operation. A proof irrelevance result showing $\mathrm{trm}(\Gamma, A, \mathcal{F}) = \mathrm{trm}(\Gamma, A, \mathcal{F}')$ for any two derivations $\mathcal{F}$ and $\mathcal{F}'$ of $\Gamma \Vdash A$ then allows us to write $\Gamma \Vdash t : A$ whenever $t \in \mathrm{trm}(\Gamma, A, \mathcal{F})$.

Since subject reduction is immediate, soundness is also immediate from the definition. Completeness follows from the usual fundamental theorem of logical relations (long proof).

Lemma 2 (Soundness)**.** *If* $\Gamma \Vdash J$ *then* $\Gamma \vdash J$*.*

Theorem 3 (Completeness)**.** *If* $\Gamma \vdash J$ *then* $\Gamma \Vdash J$*.*

Corollary 4 (Canonicity)**.** *If* $\vdash t : \mathsf{N}$ *then* $\vdash t \to^* \mathsf{S}^k\,0 : \mathsf{N}$ *for some* k*.*

Corollary 5 (The function type constructor Π is injective)**.** *If* $\Gamma \vdash \Pi(x{:}F)G = \Pi(x{:}F')G'$ *then* $\Gamma \vdash F = F'$ *and* $\Gamma, x{:}F \vdash G = G'$*.*

We then define algorithmic equality $\Gamma \vdash A\ \mathsf{conv}\ B$ and $\Gamma \vdash a\ \mathsf{conv}\ b : A$. Intuitively it says that two terms are convertible if the have a common normal form. Soundness and conversion have simple inductive proofs, thanks to the meta theory established with the logical relation.

Lemma 6 (Soundness of algorithmic equality)**.** *If* $\Gamma \vdash A\ \mathsf{conv}\ B$ *then* $\Gamma \vdash A = B$ *and if* $\Gamma \vdash a\ \mathsf{conv}\ b : A$ *then* $\Gamma \vdash a = b : A$*.*

Lemma 7 (Conversion is decidable)**.** *If* $\Gamma \vdash A$ *and* $\Gamma \vdash B$ *then* $\Gamma \vdash A\ \mathsf{conv}\ B$ *is decidable. If* $\Gamma \vdash a : A$ *and* $\Gamma \vdash b : A$ *then* $\Gamma \vdash a\ \mathsf{conv}\ b : A$ *is decidable.*

On top of algorithmic equality we define a second logical relation similarly to Definition 1. The one major difference is that for two neutral terms k and l of some base type, say N, to satisfy $\Gamma \Vdash k = l : \mathsf{N}$ they need not only to be judgmentally equal but also convertible. After proving the fundamental theorem for this second logical relation, we get completeness of algorithmic equality.

Lemma 8. *If* $\Gamma \vdash A = B$ *then* $\Gamma \vdash A\ \mathsf{conv}\ B$ *and if* $\Gamma \vdash a = b : A$ *then* $\Gamma \vdash a\ \mathsf{conv}\ b : A$*.*

Theorem 9. *In type theory, judgemental equality is decidable.*

References

[1] A. Abel and G. Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1):1–36, 2012. TYPES'10 special issue.

An Extension of Martin-Löf Type Theory with Sized Types

Andreas Abel[1] and Théo Winterhalter[2]

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

Abstract

We present a dependent type theory for which termination checking is entirely type-based, through the use of sized types. Sizes are absent from terms to ensure they are irrelevant for computation and reasoning. The novelty of our approach is the combination of first class size quantification $\forall\, i \to T$ and dependent types, justified by a predicative semantics.

Proof assistants based on dependent types, such as the very successful implementation Coq [9], rely on a termination checker to validate proofs by induction, which are represented as purely functional, recursive programs. The prevailing structural termination checkers have the main drawback that they lack compositionality, i. e., one cannot abstract out arbitrary parts of a program without leaving the termination checker clueless.

Type-based termination – suggested for functional programming [8] and dependent type theory [6, 3] already two decades ago – inherits the compositionality of polymorphic type systems for termination checking. Sized types allow to encode size-change behavior of functions in their types, and this refined type signature can be used for termination checking later without referring to the code of these functions. This way, the termination checker does not need to inline code and break abstraction barriers, and it works well with abstract and modularized developments. Following experiments with the prototype MiniAgda [1], the proof assistant Agda [2] is the first practical system utilizing sized types for termination checking.

However, in the presence of dependent types, size witnesses in terms may get in the way of equality, preventing the user from proving expected properties about their programs. Consider the function gscale on sized natural numbers, which is a generalization of multiplication and division. For instance, gscale (subtract 1) (add 2) implements scaling by $\frac{3}{2}$. It can be implemented in Agda using first-class size polymorphism for argument f. Note that $\uparrow$ is the size successor. The function scale1, where both f and g are the identity, should behave as the identity function.

```
data Nat : (i : Size) → Set where
  zero : ∀ i              → Nat (↑ i)
  suc  : ∀ i (n : Nat i)  → Nat (↑ i)

gscale : (∀ i → Nat i → Nat i) → (Nat ∞ → Nat ∞) → ∀ i → Nat i → Nat ∞
gscale f g .(↑ j) (zero j)   = zero ∞
gscale f g .(↑ j) (suc j n)  = g (suc ∞ (gscale f g j (f j n)))

scale1 = gscale (λ _ x → x) (λ x → x)
```

However, in a proof of scale1 $i\ n \equiv n$ by induction on n we get stuck in both cases, since the size arguments on the constructors zero and suc are not unifiable ($\infty \neq j$). In the following proof skeleton, we only show the goal and its reduced form for both cases.

```
scale1-id : ∀ i (n : Nat i) → scale1 i n ≡ n
scale1-id .(↑ j) (zero j)   = goal (scale1 (↑ j) (zero j) ≡ zero j)
                              goal (zero ∞ ≡ zero j)
scale1-id .(↑ j) (suc j n)  = goal (scale1 (↑ j) (suc j n) ≡ suc j n)
                              goal (suc ∞ (scale1 j n) ≡ suc j n)
```

We aspire a language where sizes are irrelevant for computation and definitional equality. The purpose of this work is to exhibit a calculus with size annotations omitted at the term level and only present on the type level, handling first-class quantification by subtyping.

This work consists of the definition of such a system along with a bidirectional algorithm for type checking and an algorithmic equality directed by size-erased types. The language we present extends Martin-Löf Type Theory by sized natural numbers, case distinction, and size-based recursion over natural numbers. In our effort to make sizes irrelevant, size abstraction and application is silent in terms which means there are no size arguments in terms that could get in the way of equality. Unlike previous works on sized dependent types [5, 4, 7, 10], we permit arbitrary rank (not only ML-style) size quantification $\forall\, i \to T$ in types.

We provide algorithms for evaluation and conversion checking. The use of sized-erased terms in algorithmic equality allows it to be syntax-directed for inferable terms.

We introduce a logical relation indexed by environments of ordinals to deduce normalization and subject reduction and eventually soundness of conversion checking. We then continue to prove its completeness and termination. Finally, we are able to derive a bidirectional type checker, assuming we have an algorithm to guess sizes verifying linear constraints.

References

[1] Andreas Abel. MiniAgda: Integrating sized and dependent types. In *Partiality and Recursion (PAR 2010)*, volume 43 of *EPTCS*, pages 14–28, 2010.

[2] AgdaTeam. The Agda Wiki, 2015.

[3] Roberto M. Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory (extended abstract). In *FoSSaCS'98*, volume 1378 of *LNCS*, pages 48–62. Springer, 1998.

[4] 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.

[5] 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.

[6] Eduardo Giménez. Structural recursive definitions in type theory. In *ICALP'98*, volume 1443 of *LNCS*, pages 397–408. Springer, 1998.

[7] Benjamin Grégoire and Jorge Luis Sacchini. On strong normalization of the calculus of constructions with type-based termination. In *LPAR'10*, volume 6397 of *LNCS*, pages 333–347. Springer, 2010.

[8] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL'96*, pages 410–423. ACM, 1996.

[9] INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.5 edition, 2016.

[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 Strongly Normalizing Computation Rule for Univalence in Higher-Order Propositional Logic

Robin Adams[1], Marc Bezem[1], and Thierry Coquard[2]

[1] Universitetet i Bergen, Bergen, Norway
{robin.adams,marc}@uib.no
[2] University of Gothenburg, Gothenburg, Sweden
coquand@chalmers.se

Homotopy type theory offers the promise of a formal system for the univalent foundations of mathematics. However, if we simply add the univalence axiom to type theory, then we lose the property of canonicity — that every term computes to a normal form. A computation becomes 'stuck' when it reaches the point that it needs to evaluate a proof term that is an application of the univalence axiom. So we wish to find a way to compute with the univalence axiom.

As a first step, we present here a system of higher-order propositional logic, with a universe Ω of propositions closed under implication and quantification over any simple type over Ω. We add a type $M =_A N$ for any terms M, N of type A, and two ways to prove an equality: reflexivity, and the univalence axiom. We present reduction relations for this system, and prove the reduction confluent and strongly normalizing on the well-typed terms.

We have begun to formalize this proof in Agda, and intend to complete the formalization by the date of the workshop.

Predicative higher-order propositional logic with equality. We call the following type theory *predicative higher-order propositional logic*. It contains a universe Ω of propositions that contains $\bot$ and is closed under implication $\supset$. The system also includes the higher-order types that can be built from Ω by $\to$. Its grammar and rules of deduction are as follows.

$$\begin{array}{llcl} \text{Proof} & \delta & ::= & p \mid \delta \cdot \delta \mid \lambda p : \phi.\delta \\ \text{Term} & M, \phi & ::= & x \mid \bot \mid MM \mid \lambda x : A.M \mid \phi \supset \phi \\ \text{Type} & A & ::= & \Omega \mid A \to A \end{array}$$

$$\frac{}{\langle\rangle \text{ valid}} \qquad \frac{\Gamma \text{ valid}}{\Gamma, x : A \text{ valid}} \qquad \frac{\Gamma \vdash \phi : \Omega}{\Gamma, p : \phi \text{ valid}} \qquad \frac{\Gamma \text{ valid}}{\Gamma \vdash x : A}\ (x : A \in \Gamma) \qquad \frac{\Gamma \text{ valid}}{\Gamma \vdash p : \phi}\ (p : \phi \in \Gamma)$$

$$\frac{\Gamma \text{ valid}}{\Gamma \vdash \bot : \Omega} \qquad \frac{\Gamma \vdash \phi : \Omega \quad \Gamma \vdash \psi : \Omega}{\Gamma \vdash \phi \supset \psi : \Omega}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad \frac{\Gamma \vdash \delta : \phi \supset \psi \quad \Gamma \vdash \epsilon : \phi}{\Gamma \vdash \delta \cdot \epsilon : \psi}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma, p : \phi \vdash \delta : \psi}{\Gamma \vdash \lambda p : \phi.\delta : \phi \supset \psi} \qquad \frac{\Gamma \vdash \delta : \phi \quad \Gamma \vdash \psi : \Omega}{\Gamma \vdash \delta : \psi}\ (\phi \simeq \psi)$$

Extensional equality. On top of this system, we add an equality predicate that satisfies univalence.

$$\begin{array}{llcl} \text{Term} & M, \phi & ::= & \cdots \mid M =_A M \\ \text{Proof} & \delta & ::= & \cdots \mid \mathsf{ref}\,(M) \mid \mathsf{univ}_{\phi,\phi}\,(\delta, \delta) \mid \lambda\!\!\lambda\!\!\lambda x : x =_A x.\delta \mid \delta \supset \delta \mid \delta\delta \\ & & & \mid \delta^+ \mid \delta^- \end{array}$$

- For any $M : A$, there is an equality proof $\mathsf{ref}\,(M) : M =_A M$.
- **Univalence.** Given proofs $\delta : \phi \supset \psi$ and $\epsilon : \psi \supset \phi$, there is an equality proof $\mathsf{univ}_{\phi,\psi}\,(\delta, \epsilon) : \phi =_\Omega \psi$.
- Given a proof $\delta : \phi =_\Omega \psi$, we have proofs $\delta^+ : \phi \supset \psi$ and $\delta^- : \psi \supset \phi$.
- Given an equality proof $\Gamma, x : A, y : A, e : x =_A y \vdash \delta : Mx =_B Ny$, there is an equality proof $\Gamma \vdash \lambda\!\!\lambda\!\!\lambda e : x =_A y.\delta : M =_{A\to B} N$. (Here, e, x and y are bound within δ.)
- **Congruence.** If $\delta : \phi =_\Omega \phi'$ and $\epsilon : \psi =_\Omega \psi'$ then $\delta \supset \epsilon : \phi \supset \psi =_\Omega \phi' \supset \psi'$. If $\delta : M =_{A\to B} M'$ and $\epsilon : N =_A N'$ then $\delta\epsilon : MN =_B M'N'$.

The reduction relation. We define the following reduction relation on proofs and equality proofs.

$$(\mathsf{ref}\,(\phi))^+ \rightsquigarrow \lambda x : \phi.x \qquad (\mathsf{ref}\,(\phi))^- \rightsquigarrow \lambda x : \phi.x \qquad \mathsf{univ}_{\phi,\psi}\,(\delta,\epsilon)^+ \rightsquigarrow \delta \qquad \mathsf{univ}_{\phi,\psi}\,(\delta,\epsilon)^- \rightsquigarrow \epsilon$$

$$(\mathsf{ref}\,(\phi) \supset \mathsf{univ}_{\psi,\chi}\,(\delta,\epsilon)) \rightsquigarrow \mathsf{univ}_{\phi\supset\psi,\phi\supset\chi}\,(\lambda f : \phi \supset \psi.\lambda x : \phi.\delta(fx), \lambda g : \phi \supset \chi.\lambda x : \phi.\epsilon(gx))$$
$$(\mathsf{univ}_{\phi,\psi}\,(\delta,\epsilon) \supset \mathsf{ref}\,(\chi)) \rightsquigarrow \mathsf{univ}_{\phi\supset\chi,\psi\supset\chi}\,(\lambda f : \phi \supset \chi.\lambda x : \psi.f(\epsilon x), \lambda g : \psi \supset \chi.\lambda x : \phi.g(\delta x))$$
$$(\mathsf{univ}_{\phi,\psi}\,(\delta,\epsilon) \supset \mathsf{univ}_{\phi',\psi'}\,(\delta',\epsilon') \rightsquigarrow \mathsf{univ}_{\phi\supset\phi',\psi\supset\psi'}\,(\lambda f : \phi \supset \phi'.\lambda x : \psi.\delta'(f(\epsilon x)), \lambda g : \psi \supset \psi'.\lambda y : \phi.\epsilon'(g(\delta y)))$$

$$(\mathsf{ref}\,(\phi) \supset \mathsf{ref}\,(\psi)) \rightsquigarrow \mathsf{ref}\,(\phi \supset \psi) \qquad \mathsf{ref}\,(M)\,\mathsf{ref}\,(N) \rightsquigarrow \mathsf{ref}\,(MN)$$
$$(\mathsf{ref}\,(\lambda x : A.M))\delta \rightsquigarrow \{\delta/x\}M \qquad (\delta \text{ a normal form not of the form } \mathsf{ref}\,(\_))$$
$$(\lambda\!\!\lambda\!\!\lambda e : x =_A y.\delta)\epsilon \rightsquigarrow [M/x, N/y, \epsilon/e]\delta \qquad (\epsilon : M =_A N)$$

Here, $\{\delta/x\}M$ is an operation called *path substitution* defined such that, if $\delta : N =_A N'$, then $\{\delta/x\}M : [N/x]M = [N'/x]M$.

Main Theorem.

Theorem 1. *In the system described above, all typable terms, proofs and equality proofs are confluent and strongly normalizing. Every closed normal form of type $\phi =_\Omega \psi$ either has the form $\mathsf{ref}\,(\_)$ or $\mathsf{univ}(\_,\_)$. Every closed normal form of type $M =_{A\to B} N$ either has the form $\mathsf{ref}\,(\_)$ or is a $\lambda\!\!\lambda\!\!\lambda$-term.*

Thus, we know that a well-typed computation never gets 'stuck' at an application of the univalence axiom.

Proof. The proof uses the method of Tait-style computability. We define the set of *computable* terms $E_\Gamma(A)$ for each type A, and computable proofs $E_\Gamma(M =_A N)$ for any terms $\Gamma \vdash M, N : A$. We prove that reduction is locally confluent, and that the computability predicates are closed under reduction and well-typed expansion. We can then prove that, if $\Gamma \vdash M : A$, then $M \in E_\Gamma(A)$; and if $\Gamma \vdash \delta : M =_A N$, then $\delta \in E_\Gamma(M =_A N)$.

Remark. Tait's proof relies on confluence, which does not hold for this reduction relation in general. In the proof, we prove confluence 'on-the-fly'. That is, whenever we require a term to be confluent, the induction hypothesis provides us with the fact that that term is computable, and hence strongly normalizing and confluent.

Representing the Process Algebra CSP in Type Theory

Bashar Igried and Anton Setzer

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

Abstract

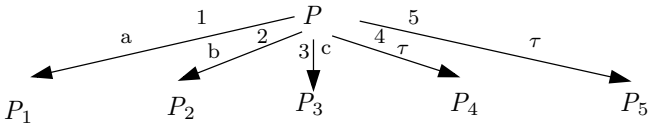
We introduce the library CSP-Agda which represents CSP processes in Agda. CSP-Agda allows to prove in Agda properties of CSP processes. CSP processes are implemented coinductively (or coalgebraically). They are formed like inductive data types from atomic operations, but infinite loops, i.e. non-wellfounded processes, are allowed.

1 Introduction

Mathematical induction is a backbone of programming and program verification [3]. Many data structures can be defined as inductive data types, also called algebraic data types. This allows to define programs using them by recursion and to prove properties by induction. In theorem proving elements of inductive types are well-founded, which means that if we consider an element of an inductive data type as a tree, there are no infinite branches. In programming one has to deal as well with potentially non-terminating programs, which corresponds to trees with infinite branches, i.e. non-well-founded data types. They are provided by the dual of inductive types, coinductive data types, also called coalgebras. Coinductive data types can be used as data types, for defining semantics, we can program with them using corecursion or guarded recursion, and reason about them using coinduction. In this paper, we will represent the process algebra CSP in a coinductive form in dependent type theory, and implement it in the Agda. Our approach is inspired by the representation of interactive programs in Agda by Peter Hancock and the second author. Since Agda is an interactive theorem prover, this allows to reason about CSP processes and prove the correctness of programs.

2 Representing CSP Processes in Agda

Processes in our approach are similar to interactive programs. Processes are defined using an atomic operation, which defines a new process by determining the transitions it can make together with the processes we obtain when firing these transitions. Processes can loop, therefore they are defined coinductively, using the representation of coalgebras as record types in Agda. The standard operations for forming processes, such as external choice or internal choice, are defined rather than considered as atomic as in process algebras. When defining processes recursively, we require them to be productive, which means for a process we can determine which next transitions it can make and the next processes after firing these transitions. The termination checker of Agda will check whether elements of coalgebraic data types are productive. Our approach is based on the algebraic nature of CSP language and on the operational characterisation of the behavioural semantics.



A CSP process is given by the labelled and silent transitions it can make. Labelled transitions correspond to external choice and silent transitions to internal choice. So we have in case of a process

progressing (1) an index set E of external choices and for each external choice e a Label ($Lab\ e$) and a next process ($PE\ e$); and (2) an index set of internal choices I and for each internal choice i a next process ($PI\ i$). In mathematical notation a process is represented as follows (assuming a set of labels $\mathsf{Label} : \mathsf{Set}$):

$$\begin{aligned}\mathsf{Process} = \{\mathsf{node}(E, Lab, PE, I, PI) \mid\ & E \in \mathsf{Choice}, \quad Lab : E \to \mathsf{Label}, \quad PE : E \to \mathsf{Process}, \\ & I \in \mathsf{Choice}, \quad PI : I \to \mathsf{Process}\ \}\end{aligned}$$

The set of choice sets is given in Agda as a universe. The operations from CSP are given as defined operations. We give here only the simplest operation, the interleaving of two processes. Assuming $P_i = \mathsf{node}(E_i, Lab_i, PE_i, I_i, PI_i)$, we define the interleaving $P_1 \ ||| \ P_2 = \mathsf{node}(E, Lab, PE, I, PI)$ by $E = E_1 + E_2$, $Lab\ (\mathsf{inl}\ e) = Lab_1\ e$, $Lab\ (\mathsf{inr}\ e) = Lab_2\ e$, $PE\ (\mathsf{inl}\ e) = (PE_1\ e) \ ||| \ P_2$, $PE\ (\mathsf{inr}\ e) = P_1 \ ||| \ (PE_2\ e)$, similarly for I and PI.

The other operations (external choice, internal choice, parallel operations, hiding, renaming, etc.) are defined in a similar way. Next we define traces model and refinement and show standard laws for processes module trace equivalence, for instance $P \ ||| \ Q$ is equivalent to $Q \ ||| \ P$.

3 Related Work

There have been several successful attempts of combining functional programming with CSP. Brown introduced in [1] a library (CHP) in Haskell. Since Haskell lacks explicit support for concurrency, they used a Haskell monad to provide a way to explicitly specify and control sequence and effects. Fontaine [2] gave another successful attempt of implementing the operational semantics of CSP as presented in [6] using the functional programming language Haskell. He presented a new tool for animation and model checking for CSP. Fontaine used a monad in order to model Input/Output, partial functions, state, non-determinism, monadic parser and passing of an environment. Lopez et. al. [4] gave further examples of combining functional programming with process algebras. They used the functional program Eden in order to translate VPSPA specification into Eden programs. Mossakowski et. al. in [5] gave a good example of using coalgebras in order to extend the specification language CASL.

References

[1] Neil C. C. Brown. Communicating Haskell processes: Composable explicit concurrency using monads. In *The thirty-first Communicating Process Architectures Conference, CPA 2008, organised under the auspices of WoTUG and the Department of Computer Science of the University of York, York, Yorkshire, UK, 7-10 September 2008*, pages 67–83, 2008.

[2] Marc Fontaine. *A Model Checker for CSP-M.* PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2011.

[3] K. Rustan M. Leino and Michal Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 382–398, 2014.

[4] Natalia López, Manuel Núñez, and Fernando Rubio. Stochastic process algebras meet Eden. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 29–48, London, UK, UK, 2002. Springer-Verlag.

[5] Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-coalgebraic specification in cocasl. *J. Log. Algebr. Program.*, 67(1-2):146–197, 2006.

[6] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

Higher Categories in Homotopy Type Theory

Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus

University of Nottingham

As homotopy type theory is viewed as a possible foundation of mathematics, it is natural to ask whether it can be used to develop category theory. This question has already been considered frequently. As we know, ordinary category theory can indeed be done in a nice way [1]. Given that types carry the structure of ∞-groupoids, it is not surprising that ordinary categories are often not sufficient. In particular, the universe itself is *not* a category in the sense of [1]. Therefore, what we want is a theory of $(\infty, 1)$-categories (simply referred to as ∞-*categories*). This is partially explored by Cranch [4], however only *concrete* categories (whose higher structure is reflected in the universe) are covered.

We expect that a general theory of ∞-categories has many applications in homotopy type theory. As equalities stated internally carry structure, different equalities are a priori not necessarily *coherent*. More often than not, this coherence will however be necessary for further constructions. While it is sometimes possible, it often does not seem feasible to handle a potentially huge number of coherence conditions manually. An example and main motivation for our development is the the project that develops a syntactical theory of higher inductive types, pursued by Dijkstra, Nordvall Forsberg, and two of the current authors [2] and is based on the semantics for higher inductive types described by Lumsdaine and Shulman [7]. The authors work with generalised containers, container algebras, and algebra morphisms, but the presentation is rendered extremely cumbersome by the fact that all the categorical laws only hold up to homotopy. For any given representation of a higher inductive type, stated as a list of constructors, the number of coherences that need to be considered is finite. In principle, this should allow the construction to go through; however, in practice, the sheer amount of these coherences cannot be handled manually in all but the most trivial cases. With a proper framework for ∞-categories, we expect that we get a clean way of resolving this problem.

A standard model of ∞-categories in set theory are Kan simplicial sets. This is essentially the notion of ∞-categories that we want to use, replacing sets by types. Note that we do not crucially insist on having a *type* of ∞-categories. Unless we extend homotopy type theory by some "infinitary" construction, this would in fact be an unreasonable expectation akin to the famous open problem of defining semisimplicial types. This is an important difference compared to the development of categories in the sense of [1]. What we want to settle for instead is a more "external" notion of ∞-category which will however have a *type* of cells on any given level. A framework in which this can be formalised would be Voevodsky's homotopy type system [9] or our 2-level theory [3]. Not having a type of ∞-categories does not seem to be a problem for typical applications. For example in the described project of developing a syntactical theory of higher inductive types, it will be sufficient to extract a *finite* number of coherence conditions from the ∞-categorical considerations; and such a finite collection will form a type.

As a further simplification, we choose to drop the requirement of degeneracies (or identities). This seems to be fine for the application of handling coherences (in further work, we will also investigate the possibility to add degeneracies in the way presented by Harpaz [5]). The advantage is that we can then consider type-valued contravariant diagrams over the *direct* category Δ_+, the category of finite ordinals and strictly increasing functions. In particular, we can consider Reedy fibrant diagrams, which can be described inductively (see [8]). This also ensures that it is reasonable to ask for *strict* semisimplicial laws.

In detail, let us write Type for the (strict, non-internally formulated) category of types and functions. Let $D : \Delta_+^{\mathrm{op}} \to \mathsf{Type}$ be a Reedy fibrant functor. Define $\mathsf{Sp}^D : \Delta_+^{\mathrm{op}} \to \mathsf{Type}$ to be the

nerve or *spine* functor of D, with $\mathsf{Sp}^D_n :\equiv D_1 \times_{D_0} \ldots \times_{D_0} D_1$. Borrowing the usual terminology, we can say that D is a *semi-Segal type* if the canonical fibrations $D_n \twoheadrightarrow \mathsf{Sp}^D_n$ are all equivalences. We can then show that D is a semi-quasicategory if and only if it is a semi-Segal type (essentially by the same argument as exhibited in [6]).

Let us outline some examples of ∞-semicategories. First, consider any type A, and the diagram $\Delta_+^{\mathrm{op}} \to \mathsf{Type}$ which is constantly A. In [6], a fibrant replacement of this diagram is constructed explicitly. This yields indeed an ∞-semicategory, called the *equality semisimplicial type* $\mathcal{E}A$ in [6]. Not surprisingly, it fulfils the stronger property of being an ∞-*semigroupoid*.

Second, consider the family $D : \mathbb{N} \to \mathsf{Type}$, with $D_0 :\equiv \mathcal{U}$ (i.e. elements are small types), $D_1 :\equiv \Sigma\,(X_0, X_1 : \mathcal{U})\,.\,X_0 \to X_1$ (pairs of types and a function between them), $D_2 :\equiv \Sigma\,(X_0, X_1, X_2 : \mathcal{U})\,.\,(X_0 \to X_1) \times (X_1 \to X_2)$ (a chain $X_0 \to X_1 \to X_2$ of types), and so on; in general, D_n are chains of length n. D can be completed to a functor $\Delta_+^{\mathrm{op}} \to \mathsf{Type}$. Exactly as before, we can then take a Reedy fibrant replacement of this functor which, by construction, is a semi-Segal type. It corresponds to the ∞-semicategory of types and we call it TYPE.

In the sketched situation, we are lucky: completing the family $D : \mathbb{N} \to \mathsf{Type}$ to an actual functor from Δ_+^{op} is straightfoward as associativity of function composition holds strictly. Unfortunately, not all ∞-semicategories of interest can be dealt with in this fashion, since a similar construction would yield a family of types D that cannot be regarded as a strict functor in any obvious way. This already happens in the case of *pointed types*, where the analogous approach would be to start with chains of pointed types and pointed maps. However, we can instead consider chains of types and functions (as before) and a point only in the *first* type of the chain. This is an equivalent representation which carries a strict structure, giving rise to the ∞-semicategory PTYPE.

From the basic ingredients TYPE and PTYPE, and the fibration $\mathsf{PTYPE} \twoheadrightarrow \mathsf{TYPE}$, we can construct more sophisticated examples. The simplest interesting example is taking the local exponential of PTYPE with itself in context TYPE, which leads essentially to algebras over the identify functor. This way, we hope to achieve a reasonably theory of ∞-semicategories in homotopy type theory which can then also be used to treat coherences in a principled way.

References

[1] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science (MSCS)*, pages 1–30, Jan 2015.

[2] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. Towards a theory of higher inductive types. Presentation at TYPES'15, Tallinn, Estonia, 20 May 2015.

[3] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending homotopy type theory with strict equality. *ArXiv e-prints*, April 2016.

[4] James Cranch. Concrete categories in homotopy type theory. *ArXiv e-prints*, Nov 2013.

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

[6] Nicolai Kraus. The general universal property of the propositional truncation. In *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, *LIPIcs* vol. 39, pages 111–145, 2015.

[7] Peter LeFanu Lumsdaine and Michael Shulman. Semantics of higher inductive types, 2013. Unpublished note, `ncatlab.org/homotopytypetheory/files/hit-semantics.pdf`.

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

[9] Vladimir Voevodsky. A simple type system with two identity types, 2013. Unpublished note.

Partiality, Revisited

Thorsten Altenkirch[1*] and Nils Anders Danielsson[2†]

[1] School of Computer Science, University of Nottingham, UK
[2] University of Gothenburg and Chalmers University of Technology, Sweden

This note presents work in progress on representing partial computations in type theory. We define a monad $-_\perp : \mathbf{Set} \to \mathbf{Set}$.[1] This monad is a pointed ω-CPO, and can be used to solve recursive equations: given an ω-continuous function $f : (A \to B_\perp) \to (A \to B_\perp)$ we can construct a function $\mathit{fix}(f) : A \to B_\perp$ satisfying $\mathit{fix}(f) = f(\mathit{fix}(f))$.

Capretta (2005) gave a similar construction using setoids, but setoids are arguably awkward to work with, so our aim is to avoid them. A natural idea is to use quotient types, as suggested by Capretta et al. (2005) and worked out in more detail by Chapman et al. (2015): First define the delay monad $\mathit{Delay} : \mathbf{Set} \to \mathbf{Set}$ coinductively by the constructors $\eta : A \to \mathit{Delay}(A)$ and $\mathit{later} : \mathit{Delay}(A) \to \mathit{Delay}(A)$. Then one can, for instance, define the least value $\perp : \mathit{Delay}(A)$ as the solution to the guarded equation $\perp = \mathit{later}(\perp)$. However, the delay monad distinguishes computations that proceed with different speed (different number of *later* constructors). This can be remedied by quotienting the delay monad by an equivalence relation corresponding to weak bisimilarity. There are a number of ways to define this relation, for instance the following one (Capretta, 2005): First define a relation $\downarrow : A_\perp \to A \to \mathbf{Prop}$, where $p \downarrow a$ means that p terminates with the value a, inductively by $\eta(a) \downarrow a$ and $p \downarrow a \to \mathit{later}(p) \downarrow a$. Weak bisimilarity $\approx : \mathit{Delay}(A) \to \mathit{Delay}(A) \to \mathbf{Prop}$ can then be defined by $p \approx q := \Pi_{a:A}(p \downarrow a \leftrightarrow q \downarrow a)$, and $A_\perp$ as the quotient type $\mathit{Delay}(A)/{\approx}$. However, Chapman et al. (2015) noticed a potential problem with this construction: it seems to be hard or impossible to prove that $-_\perp$ is a monad in (some variant of) type theory. They also showed that $-_\perp$ really is a monad under the assumptions of countable choice and propositional extensionality.

Our aim is to show that a partiality monad can be constructed without having to introduce countable choice. We observe, as did Chapman et al. (2015), that the situation is reminiscent of the situation with the Cauchy reals. If the Cauchy reals are defined as a quotient, then it is for instance impossible to prove a specific form of the statement that every Cauchy sequence of Cauchy reals has a limit using $\mathsf{IZF}_{\mathrm{Ref}}$, a constructive set theory without countable choice (Lubarsky, 2007). The Univalent Foundations Program (2013, Section 11.3) circumvents this problem by defining the Cauchy reals as "the free complete metric space generated by $\mathbb{Q}$", using a higher inductive-inductive type that mutually defines the real numbers—including an inclusion of rational numbers and a limit construction—and a certain relation. We note that this definition of the real numbers forms a set and that the relation is propositional, so the higher inductive-inductive type used is a *quotient inductive-inductive type* (Altenkirch and Kaposi, 2016).

Using a similar approach we mutually define $A_\perp : \mathbf{Set}$ and $\sqsubseteq : A_\perp \to A_\perp \to \mathbf{Prop}$, where $\sqsubseteq$ represents information ordering, as a quotient inductive-inductive type:

$$\begin{array}{rcl} \perp & : & A_\perp \\ \eta & : & A \to A_\perp \\ \bigsqcup & : & \Pi_{f:\mathbb{N}\to A_\perp}(\Pi_{n:\mathbb{N}} f(n) \sqsubseteq f(n+1)) \to A_\perp \end{array}$$

*Supported by EPSRC grant EP/M016951/1 and USAF grant FA9550-16-1-0029.
†Supported by a grant from the Swedish Research Council (621-2013-4879).

[1] For the purpose of this note we define **Set** (**Prop**) as the type of types in the first universe that are sets (propositions) in the sense of the Univalent Foundations Program (2013).

To improve readability we present the constructors for $\sqsubseteq$ using inference rules:[2]

$$\frac{}{d \sqsubseteq d} \qquad \frac{}{\bot \sqsubseteq d} \qquad \frac{\bigsqcup(f,p) \sqsubseteq d}{\Pi_{n:\mathbb{N}} f(n) \sqsubseteq d} \qquad \frac{\Pi_{n:\mathbb{N}} f(n) \sqsubseteq d}{\bigsqcup(f,p) \sqsubseteq d}$$

These rules say that the relation is reflexive with $\bot$ as the least element, and that $\bigsqcup(f,p)$ has a given upper bound iff the sequence f has the same upper bound. We add two constructors for equality, one which turns $\sqsubseteq$ into a partial order (transitivity can be proved), and one which makes $\sqsubseteq$ propositional (one can then prove that $A_\bot$ is a set):

$$\frac{d \sqsubseteq d' \qquad d' \sqsubseteq d}{d = d'} \qquad \frac{p, q : d \sqsubseteq d'}{p = q}$$

We have showed—without assuming countable choice—that $-_\bot$ is a monad. The construction and this proof have been implemented in Agda,[3] using an experimental rewriting feature developed by Andreas Abel and Jesper Cockx to support higher inductive-inductive types.

However, we have not yet verified that this monad is correctly defined. Together with Paolo Capriotti and Nicolai Kraus we have ruled out the risk that the monad is trivial by proving that $\bot \neq \eta(x)$ (in Agda, using the univalence axiom), but we have not yet established a firm connection between the construction presented here and the quotiented delay monad mentioned above.

Further experiments might reveal whether our construction provides a good basis for the development of a theory of partial functions within type theory.

References

Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *POPL'16, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 18–29, 2016. doi:10.1145/2837614.2837638.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–28, 2005. doi:10.2168/LMCS-1(2:1)2005.

Venanzio Capretta, Thorsten Altenkirch, and Tarmo Uustalu. Partiality is an effect. Slides for a talk given by Uustalu at the 22nd meeting of IFIP Working Group 2.8, 2005.

James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the delay monad by weak bisimilarity. In *Theoretical Aspects of Computing – ICTAC 2015*, volume 9399 of *LNCS*, pages 110–125. 2015. doi:10.1007/978-3-319-25150-9_8.

Robert S. Lubarsky. On the Cauchy completeness of the constructive Cauchy reals. *Mathematical Logic Quarterly*, 53(4–5):396–414, 2007. doi:10.1002/malq.200710007.

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

[2] Because $\sqsubseteq$ is propositional we omit the constructor names.

[3] With the K rule turned off, and with minor differences from this presentation. The source code can at the time of writing be found via Danielsson's personal web page (`http://www.cse.chalmers.se/~nad/`).

Normalisation by Evaluation for Dependent Types*

Thorsten Altenkirch and Ambrus Kaposi

University of Nottingham, United Kingdom
{txa,auk}@cs.nott.ac.uk

Abstract

We prove normalisation for a basic dependent type theory using the technique of normalisation by evaluation (NBE). NBE works by evaluating the syntax into a model and then computing normal forms by quoting semantic objects into normal terms. As model we use the syntax glued together with a proof-relevant logical predicate. The syntax is defined internally in type theory as a quotient-inductive type. We use a typed presentation of the syntax, we don't mention preterms. Parts of the construction are formalised in Agda. A full paper will be presented at FSCD 2016.

Specifying normalisation

We denote the type of well typed terms of type A in context Γ by $\mathsf{Tm}\,\Gamma\,A$. This type is defined as a quotient inductive inductive type (QIIT, see [3]): in addition to normal constructors for terms such as lam and app, it also has equality constructors e.g. expressing the β computation rule for functions. An equality $t \equiv_{\mathsf{Tm}\,\Gamma\,A} t'$ expresses that t and t' are convertible. Typed normal forms are denoted $\mathsf{Nf}\,\Gamma\,A$ and are defined mutually with neutral terms $\mathsf{Ne}\,\Gamma\,A$ and the embedding $\ulcorner - \urcorner : \mathsf{Nf}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A$. Normalisation is given by a function norm which is an isomorphism[1]:

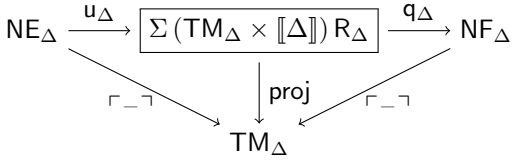
$$\text{completeness}\circlearrowleft \qquad \mathsf{norm}\downarrow \quad \frac{\mathsf{Tm}\,\Gamma\,A}{\mathsf{Nf}\,\Gamma\,A} \quad \uparrow\ulcorner - \urcorner \qquad \circlearrowright\text{stability}$$

Completeness says that normalising a term produces a term which is convertible to it: $t \equiv \ulcorner \mathsf{norm}\,t \urcorner$. Stability expresses that there is no redundancy in the type of normal forms: $n \equiv \mathsf{norm}\ulcorner n \urcorner$. The usual notion of soundness of the semantics, that is, if $t \equiv t'$ then $\mathsf{norm}\,t \equiv \mathsf{norm}\,t'$ is given by congruence of equality. The elimination rule for the QIIT of the syntax ensures that every function defined from the syntax respects the equality constructors.

NBE for simple types

NBE is one way to implement the above specification. It works by evaluating the syntax in a model and defining a quote function which turns semantic objects into normal terms. We follow the categorical approach to NBE as given by [2] for simple types. Here the model is a presheaf model over the category of renamings (objects are contexts, morphisms are lists of variables) and the interpretation of the base type is the set of normal terms at the base type.

The structure of the normalisation proof is given by the following diagram. It summarizes normalisation of a substitution into context Δ, a similar diagram can be given for terms.



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

[1] This might by surprising however it can be explained by the fact that the conversion relation is part of the equality structure on terms.

NE_Δ, TM_Δ and NF_Δ denote the Yoneda embeddings of neutral terms, terms and normal terms, respectively. These are all presheaves over the category of renamings where the action on objects returns lists of neutral terms Nes, substitutions Tms and lists of normal forms Nfs, respectively.

$$\mathsf{NE}_\Delta\,\Gamma := \mathsf{Nes}\,\Gamma\,\Delta \qquad \mathsf{TM}_\Delta\,\Gamma := \mathsf{Tms}\,\Gamma\,\Delta \qquad \mathsf{NF}_\Delta\,\Gamma := \mathsf{Nfs}\,\Gamma\,\Delta$$

The presheaf interpretation of the context Δ is denoted $[\![\Delta]\!]$. R_Δ denotes a binary logical relation at context Δ. This is a relation between the syntax TM and the presheaf semantics $[\![-]\!]$ and is equality at the base type. u_Δ denotes the unquote natural transformation and q_Δ is quotation. These are defined mutually by induction on contexts and types.

Normalisation of a substitution σ is given by evaluating it in the presheaf model $[\![\sigma]\!]$ and then using quote. It also needs the semantic counterpart of the identity substitution which is given by unquoting the identity substitution $\mathsf{u}_\Gamma\,\mathsf{id}$ and a witness of the logical relation which is given by the fundamental theorem for the logical relation R_σ.

$$\mathsf{norm}_\Delta\,(\sigma : \mathsf{TM}_\Delta\,\Gamma) : \mathsf{NF}_\Delta\,\Gamma := \mathsf{q}_\Delta\,\big(\sigma, [\![\sigma]\!]\,(\mathsf{u}_\Gamma\,\mathsf{id}), \mathsf{R}_\sigma\,(\mathsf{u}_\Gamma\,\mathsf{id})\big)$$

Completeness is given by commutativity of the right hand triangle. Stability can be proven by mutual induction on terms and normal forms.

NBE for dependent types

NBE has been extended to dependent types using untyped realizers [1] and a typed version has been given by [4] however without a proof of soundness. Our goal was to extend the categorical approach summarized in the previous section to dependent types. The straightforward generalisation does not work because there seems to be no way of defining unquote for Π. Here we need to define a semantic function which works for arbitrary inputs, not only those which are related to a term. It seems that we need to restrict the presheaf model to only contain such functions.

We solve this problem by merging the presheaf model $[\![-]\!]$ and the logical relation R into a proof-relevant logical predicate P. That is, we replace the boxed part of the above diagram by $\boxed{\Sigma\,\mathsf{TM}_\Delta\,\mathsf{P}_\Delta}$. In the presheaf model, the interpretation of the base type were normal forms of the base type, and the logical relation at the base type was equality of the term and the normal form (equality was proof irrelevant, hence the logical relation was proof irrelevant). In our case, the logical predicate at the base type says that there exists a normal form which is equal to the term, hence it needs to be proof relevant. It can be seen as an instance of categorical glueing.

References

[1] Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity.* PhD thesis, Habilitation, Ludwig-Maximilians-Universität München, 2013.

[2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, LNCS 953, pages 182–199, 1995.

[3] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 18–29, New York, NY, USA, 2016. ACM.

[4] Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs*, pages 93–109. Springer, 2006.

Type Inference for Ratio Control Multiset-Based Systems*

Bogdan Aman and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science, Iaşi, Romania
baman@iit.tuiasi.ro, gabriel@info.uaic.ro

Outline. The paper presents a hierarchical nested system together with a multiset-based type system involving a ratio control of the resources. This ratio control keeps the resources between lower and upper thresholds. According to a typed semantics, each rule of the system can be applied only if the left-hand side term of the rule is well-typed. A type inference is defined for deducing the type of each term of such a system. Soundness and completeness results are proved for this type inference.

Ecological stoichiometry is the study of the balance of energy and multiple elements in ecological interactions [10]. From the stoichiometric point of view, both quantity and quality need to be implicitly or explicitly modelled in producer-consumer interactions. Simple phenomenological two-dimensional producer-consumer models are mathematically tractable by extensions of the standard theory of predator-prey interactions described by Lotka and Volterra. The Lotka-Volterra model extended with the ratio-dependent interaction is presented in a rather recent book [2]. Usually, the ratio of (two) elements reflects the quality [11]. However, the models described by differential equations [7], do not explicitly track the quantities neither in the producer-consumer nor in the environment. In this paper we provide a discrete approach and a new quantitative (multiset-based) type system involving a ratio control of the resources.

Multiset-based formalisms are motivated by quantitative evolutions of various systems. In many biological systems a reaction takes place only if certain ratios between given thresholds are fulfilled (e.g, in sodium/potassium pump [3] and ratio-dependent predator-prey systems [6]). Several multiset rewriting systems are used to describe the dynamics of systems which involve parallelism and concurrent access to resources. Petri nets [9] and membrane systems [8] represent good examples of the kind of multiset-based formalisms mentioned in this paper.

The current paper presents a more general and improved approach starting from the type system of [1] in which we emphasise the ratio control of resources between lower and upper thresholds. In this way, we are able to capture the quantitative aspects and abstract conditions associated with correct evolutions. We provide a "two steps" description of behaviours, where the first step describes reactions in an "untyped" setting, and the second rules out some evolutions by imposing certain ratio thresholds. This approach allows to treat separately different aspects of modelling: first what are the possible transitions, and then under which circumstances can they take place. Also it facilitates a better understanding of the evolution in complex quantitative systems with lower and upper thresholds. We prove soundness and completeness results, and show that each rule of the system can be applied only if the left-hand side term of the rule is well-typed.

Let T be a finite set of basic types ranged over by t. Each object a in O is classified with an element of T; Γ denotes this classification. In general, different objects a and b can have the same basic type t (e.g, a and b can be both ions). When there is no ambiguity, the type associated with an object a is denoted by t_a. For each ordered pair of basic types (t_1, t_2), the

*The work was supported by a grant of the Romanian National Authority for Scientific Research, project number PN-II-ID-PCE-2011-3-0919.

existence of one function is assumed: $min : T \times T \to (0, \infty) \cup \{\diamond\}$. This function indicates the minimum ratio between the number of objects of basic types t_1 and t_2 that can be present inside a component. We consider that the maximum ratio between t_1 and t_2, denoted by $max(t_1, t_2)$ could be determined using the relation $min(t_1, t_2) \cdot max(t_2, t_1) = 1$. According to this relation, it is enough to use only the min function. For example, by taking the constraints $min(t_a, t_b) = 3$ and $min(t_b, t_a) = 1/5$, the number of objects of basic type t_a is larger than the number of objects of basic type t_b with a coefficient between three and five. $min(t_1, t_2) = \diamond$ tells that this function is undefined for the pair of types (t_1, t_2). Biologically speaking, the ratio between the types t_1 and t_2 is either unknown, or can be ignored.

We provide a type inference algorithm for deducing the type of each term in the multiset framework, and prove the soundness and completeness results for this type inference. We provide a typed semantics, and prove that each rule of the system can be applied only if the left-hand side term of the rule is well-typed.

A formalism that is somehow related to the multiset framework with components considered in this paper is the calculus of looping sequences. An essential difference is that multiset framework with components use multisets to describe objects on components, while calculus of looping sequences terms use words as looping sequences. There are various type systems defined for calculus of looping sequences. Our approach is related to [4], where a type system and type inference for the calculus of looping sequences is defined based on the number of elements. However, our type system and type inference approach uses ratio thresholds instead of exact numbers of elements being able to type systems more complex than in [4], providing also more flexibility given by lower and upper thresholds.

References

[1] B. Aman, G. Ciobanu. Behavioural Types Inspired by Cellular Thresholds. *Lecture Notes in Computer Science* **8368**, 1–15 (2014).

[2] R. Arditi, L.R. Ginzburg. *How Species Interact: Altering the Standard View on Trophic Ecology.* Oxford University Press (2012).

[3] D. Besozzi, G. Ciobanu. A P System Description of the Sodium-Potassium Pump. *Lecture Notes in Computer Science* **3365**, 210–223 (2005).

[4] L. Bioglio. Enumerated Type Semantics for the Calculus of Looping Sequences. *RAIRO - Theoretical Informatics and Applications* **45** (1), 35–58 (2011).

[5] P. Frisco, Gh. Păun, M.J. Pérez-Jiménez.(Eds.) *Emergence, Complexity and Computation Series* **7**. Springer (2014).

[6] S.-B. Hsu, T.-W. Hwang, Y. Kuang. A Ratio-Dependent Food Chain Model and Its Applications to Biological Control. *Mathematical Biosciences* **181**, 55–83 (2003).

[7] Y. Kuang, E. Beretta, Global Qualitative Analysis of a Ratio-Dependent Predator-Prey System. *Journal of Mathematical Analysis and Applications* **36**, 389–406 (1998).

[8] Gh. Păun, G. Rozenberg, A. Salomaa. (Eds.) *Handbook of Membrane Computing.* Oxford University Press (2010).

[9] C.A. Petri, W. Reisig. Petri Net. *Scholarpedia* **3**(4): 6477.

[10] R.W. Sterner, J.J. Elser. *Ecological Stoichiometry.* Princeton University (2002).

[11] H. Wang, Y. Kuang, I. Loladze. Dynamics of a Mechanistically Derived Stoichiometric Producer-Grazer Model. *Journal of Biological Dynamics* **2**(3), 286–296 (2008).

Type Theory based on Dependent Inductive and Coinductive Types

Henning Basold[1] and Herman Geuvers[2]

[1] Radboud University & CWI Amsterdam, h.basold@cs.ru.nl
[2] Radboud University & Technical University Eindhoven, herman@cs.ru.nl

Overview of the Talk

In this talk, we will develop a type theory that is based solely on dependent inductive and coinductive types. By this we mean that the only way to form new types is by specifying the type of their corresponding constructors or destructors, respectively. From such a specification, we get the corresponding recursion and corecursion principles. One might be tempted to think that such a theory is relatively weak as, for example, there is no function space type. However, as it turns out, the function space is definable as a coinductive type. In fact, we can encode the connectives of intuitionistic predicate logic: falsity, conjunction, disjunction, dependent function space, existential quantification, and equality. Further, well-known types like natural numbers, vectors etc. arise as well. The presented type theory is based on ideas from categorical logic that have been investigated before by the first author, and it extends Hagino's categorical data types to a dependently typed setting. By basing the type theory on concepts from category theory we maintain the duality between inductive and coinductive types.

The reduction relation on terms consists solely of a rule for recursion and a rule for corecursion. We can then derive the usual computation rules for encoded types these basic rules. This results in a type theory with a small set of rules, while still being fairly expressive. To further support the introduction of this new type theory, we prove subject reduction and strong normalisation of the reduction relation.

Why do we need another type theory, especially since Martin-Löf type theory (MLTT) or the calculus of inductive constructions (CoIC) are well-studied frameworks for intuitionistic logic? The main reason is that the existing type theories have no explicit dependent coinductive types. There is support for them in implementations like Coq, based on early ideas by Giménez, and Agda. However, both have no formal justification, and Coq's coinductive types are known to have problems (e.g. related to subject reduction). The calculus of constructions has been extended with streams in such a way that Coq's problems do not arise, but the problem of limited support remains. Just as Sacchini's work can be seen as formal justification of (parts of) Coq, the type theory we study here can be seen as formal justification for (an extension of) Agda's coinductive types.

One might argue that dependent coinductive types can be encoded through inductive types. However, it is not clear whether such an encoding gives rise to a good computation principle in an intensional type theory such as MLTT or CoIC. This becomes an issue once we try to prove propositions about terms of coinductive type.

Other reasons for considering a new type theory are of foundational interest. First, taking inductive and coinductive types as core of the type theory reduces the number of deduction rules considerably. For each type former one needs the corresponding type rule, and introduction and elimination rules. This makes for a considerable amount of rules in MLTT with W- and M-types, while our theory only has 6 relevant deduction rules. Second, it is an interesting fact that the (dependent) function space can be described as a coinductive type. This seems to be

widely accepted but we do not know of any formal treatment of this fact. Thus the presented type theory allows us to deepen our understanding of coinductive types.

Contributions Having discussed the raison d'être of this work, let us briefly mention the technical contributions. First of all, we introduce the type theory and show how important logical operators can be represented in it. We also discuss some other basic examples, including one that shows the difference to existing theories with coinductive types. Second, we show that computations of terms, given in form of a reduction relation, are meaningful, in the sense that the reduction relation preserves types (subject reduction) and that all computations are terminating (strong normalisation). Thus, under the propositions-as-types interpretation, our type theory can serve as formal framework for intuitionistic reasoning.

Related Work A major source of inspiration for the setup of our type theory is categorical logic. Especially, the use of fibrations helped a great deal in understanding how coinductive types should be treated. Another source of inspiration is the view of type theories as internal language or even free model for categories. This view is especially important in topos theory, where final coalgebras have been used as foundation for predicative, constructive set theory.

Let us briefly discuss other type theories that the present work relates to. Especially close is the copattern calculus, as there the coinductive types are also specified by the types of their destructors. However, said calculus does not have dependent types, and it is based on systems of equations to define terms, whereas the calculus in the present work is based on recursion and corecursion schemes.

To ensure strong normalisation, the copatterns have been combined with size annotations. Due to the nature of the reduction relation in these copattern-based calculi, strong normalisation also ensure productivity for coinductive types or, more generally, well-definedness. As another way to ensure productivity, guarded recursive types have been proposed and guarded recursion has been extended to dependent types. Guarded recursive types are not only applicable to strictly positive types, which we restrict to here, but also to positive and even negative types. However, it is not clear how one can include inductive types into such a type theory, which are, in the authors opinion, crucial to mathematics and computer science.

Sneak Preview

Let us briefly peek at the calculus we are going to see in the talk. An important type formation rule is that for coinductive types:

$$\frac{k = 1, \dots, n \qquad \sigma_k : \Gamma_k \rhd \Gamma \qquad \Theta, X : \Gamma \multimap * \mid \Gamma_k \vdash A_k : *}{\Theta \mid \varnothing \vdash \nu(X : \Gamma \multimap * ; \vec{\sigma}; \vec{A}) : \Gamma \multimap *}$$

Here, n is a positive natural number and each σ_k is a substitution for the variables of the context Γ by terms in context Γ_k. The notation $X : \Gamma \multimap *$ indicates a type constructor variable that can be instantiated with terms according to Γ. The judgement $\Theta, X : \Gamma \multimap * \mid \Gamma_k \vdash A_k : *$ then says that each A_k is a type with free type constructor variables in Θ extended with X, and free term variables in Γ_k. The intuition is that $\nu(X : \Gamma \multimap * ; \vec{\sigma}; \vec{A})$ has destructors that can only be applied to elements of this type instantiated with terms that match with σ_k and with output of type $A_k[\nu/X]$. These destructors are formally terms for each $k = 1, \dots, n$

$$\xi_k : (\Gamma_k, x : \nu \mathbin{@} \sigma_k) \multimap A_k[\nu/X],$$

where $\nu \mathbin{@} \sigma_k$ denotes the instantiation of the type with terms in σ_k.

On self-interpreters for Gödel's System T

Andrej Bauer

University of Ljubljana, Slovenia
Andrej.Bauer@andrej.com

In defiance of the received wisdom that a total programming language cannot have a self-interpreter, Brown and Palsberg [3] implemented a self-interpreter for System F_ω, which is a strongly normalizing typed λ-calculus and thus certainly a total language. In order to avoid the trivial self-interpreter they imposed certain constraints. I show that under the same constraints already Gödel's System T has a self-interpreter (Theorem 5). The construction is trivial, which makes one think that something is at fault with the notion of self-interpreter used by Brown and Palsberg. However, I show that there cannot be a significant improvement (Corollary 6) in the sense that the type of source codes must be at least as complex as the type of the programs they encode. I conclude by suggesting a definition of self-interpreter which is satisfied by Brown and Palsberg's interpreter but not by the one constructed in Theorem 5.

We work with the simply typed λ-calculus [2, §A.1], and in particular with Gödel's T, which is an extension of the simply typed λ-calculus with a ground type of natural numbers nat and *primitive* recursion at each type, see [2, §A.2] and [1]. It is strongly normalizing [1, §4.3] and expressive enough to manipulate syntax through Gödel encodings. We write $\mathsf{Prg}(\tau)$ for the set of all closed expressions (programs) of type τ.

Definition 1. A *typed self-interpreter* is given by a type ν of *(source) codes*, and for each type τ a *quoting function* $\ulcorner\text{-}\urcorner_\tau : \mathsf{Prg}(\tau) \to \mathsf{Prg}(\nu)$ and an *interpreter* $\mathtt{u}_\tau \in \mathsf{Prg}(\nu \to \tau)$ such that $\mathtt{u}_\tau \ulcorner e \urcorner_\tau \equiv_\beta e$ for all $e \in \mathsf{Prg}(\tau)$.

Note that the quoting functions need not be λ-definable, i.e., there may be no programs $\mathtt{q}_\tau$ such that $\ulcorner e \urcorner_\tau \equiv_\beta \mathtt{q}_\tau\ e$. The following theorem is the justification for the popular opinion that total languages do not have self-interpreters.

Theorem 2. *If a λ-calculus has a self-interpreter then it has fixed-point operators at all types.*

The theorem can be inverted: a simply typed λ-calculus with natural numbers and fixed-point operators has a self-interpreter, as was affirmed by Longley and Plotkin [4, Prop. 6].

Corollary 3. *System T does not have a self-interpreter.*

Proof. In System T successor $\mathsf{succ} : \mathsf{nat} \to \mathsf{nat}$ has no fixed points. □

The corollary holds for other kinds of calculi, as long as they posses endomaps without fixed points, which is typical of strongly normalizing calculi. To obtain a self-interpreter for System T we thus need to relax the definition of self-interpreters. The following one is fashioned after Brown and Palsberg [3]. We write $\mathfrak{n}(e)$ for the normal form of an expression e, and $\mathfrak{g}(e)$ for its Gödel code, which is a suitable encoding of e by a number. We write $\underline{n}$ for the numeral that represents $n \in \mathbb{N}$.

Definition 4. A *weak self-interpreter* is given by, for each type τ, a type of *(source) codes* $\Box\tau$, a *quoting function* $\ulcorner\text{-}\urcorner_\tau : \mathsf{Prg}(\tau) \to \mathsf{Prg}(\Box\tau)$, and an *interpreter* $\mathtt{u}_\tau : \mathsf{Prg}(\Box\tau \to \tau)$ such that $\mathtt{u}_\tau \ulcorner e \urcorner_\tau \equiv_\beta e$ for all $e \in \mathsf{Prg}(\tau)$. Such an interpreter is *strong* when for every type τ, the quoting function $\ulcorner\text{-}\urcorner_\tau$ is (1) *normal*: $\ulcorner e \urcorner_\tau$ is β-normal for all $e \in \mathsf{Prg}(\tau)$, and (2) *acceptable*: there is $\mathtt{g}_\tau : \Box\tau \to \mathsf{nat}$ such that $\mathtt{g}_\tau \ulcorner e \urcorner_\tau = \underline{\mathfrak{g}(e)}$ for all $e \in \mathsf{Prg}(\tau)$.

Normality expresses the idea that codes should be values and acceptability that the syntax of an expression is discernible from its code. Brown and Palsberg also require injectivity of the quoting function, which follows from our definition because $\mathfrak{g}$ is injective. They do not explicitly postulate acceptability, although they provide programs that extract the syntax of an expression from its code.

A strong self-interpreter cannot have a trivial quoting function $\ulcorner e \urcorner_\tau = e$ because codes must be β-normal, while injectivity of $\ulcorner \text{-} \urcorner_\tau$ prevents coding by β-normal forms $\ulcorner e \urcorner_\tau = \mathfrak{n}(e)$.

Theorem 5. *System T has a strong Brown-Palsberg self-interpreter.*

Proof. Define $\Box\tau = \mathsf{nat} \times \tau$, $\ulcorner e \urcorner_\tau = \langle \underline{\mathfrak{g}(e)}, \mathfrak{n}(e) \rangle$, $\mathtt{u}_\tau = \mathsf{snd}$, and $\mathtt{g}_\tau = \mathsf{fst}$. □

It is clear that the same proof applies to any calculus that has binary products, natural numbers, and any notion of normal form, such as System F_ω.

The proof of Theorem 5 abuses the fact that Brown-Palsberg interpreters allow codes to be as complex as the programs they encode. Theorem 2 prevents us from using a fixed type of codes, but perhaps $\Box\tau$ can at least be less complex than τ? We show that this is not possible for the standard notion of *level* defined inductively as $\mathsf{lev}(\mathsf{nat}) = 0$, $\mathsf{lev}(\sigma \times \tau) = \max(\mathsf{lev}(\sigma), \mathsf{lev}(\tau))$, and $\mathsf{lev}(\sigma \to \tau) = \max(1 + \mathsf{lev}(\sigma), \mathsf{lev}(\tau))$. That is, $\mathsf{lev}(\tau)$ gives the deepest nesting of $\to$ to the left in τ.

Theorem 6. *A weak self-interpreter for System T satisfies $\mathsf{lev}(\Box\tau) \geq \mathsf{lev}(\tau)$ for every type τ.*

The self-interpreter for F_ω given by Brown and Palsberg has important structural properties that Definition 4 fails to capture. For instance, their encoding of types commutes with substitution [3, Thm. 5.2] and is a congruence with respect to type equality [3, Thm. 5.3]. In the original work [5] on meta-circularity Pfenning and Lee called such phenomena *reflexivity*. Unfortunately they spoke of it at an informal level and did not provide a definition. A promising possibility, which thwarts the proof of Theorem 5 but allows Brown and Palsberg's construction, is to amend the definition of weak interpreters by requiring a term $\mathsf{app}_{\sigma,\tau} : \Box(\sigma \to \tau) \to \Box\sigma \to \Box\tau$ such that $\mathsf{app}_{\sigma,\tau}\ \ulcorner e_1 \urcorner_{\sigma\to\tau}\ \ulcorner e_2 \urcorner_\sigma \equiv_\beta \ulcorner e_1\ e_2 \urcorner_\tau$. This way we get conditions that correspond to the modal laws of necessity. Unfortunately, at present I do not know whether System T has a self-interpreter satisfying the above conditions *and* the acceptability condition from Definition 4.

Acknowledgment I thank Alex Simpson for helpful discussions and pointers to the literature.

References

[1] Jeremy Avigad and Solomon Feferman. Chapter V: Gödel's Functional ("Dialectica") Interpretation. In Samuel R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 337–405. Elsevier, 1998.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. College Publications, 1984.

[3] Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for F-omega. In *Principles of Programming Languages (POPL)*, January 2016.

[4] John Longley and Gordon Plotkin. Logical full abstraction and pcf. In *Tbilisi Symposium on Language, Logic and Computation. SiLLI/CSLI*, pages 333–352. SiLLI/CSLI, 1996.

[5] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89(1):137–159, 1991.

Design and Implementation of the Andromeda proof assistant

Andrej Bauer[1], Gaëtan Gilbert[2], Philipp Haselwarter[1], Matija Pretnar[1], and Christopher A. Stone[3]

[1] University of Ljubljana, Slovenia
[2] École Normale Supérieure Lyon, France
[3] Harvey Mudd College, USA

Andromeda [1] is a proof assistant for dependent type theory with equality reflection following the tradition of Edinburgh LCF [3]: (1) there is an abstract datatype of type-theoretic judgments whose values can only be constructed by a small *nucleus*, and (2) the user interacts with the nucleus by writing programs in a high-level, statically typed *Andromeda meta-language (AML)*. The only part of the system that needs to be trusted is the nucleus, which at present counts around 1800 lines of OCaml code.

The underlying type theory of Andromeda has dependent products and equality types (LCF and its descendants implement simple type theory). The rules for products are standard and include function extensionality. The terms are explicitly tagged with typing annotations, which is necessary because we want to avoid various anomalies caused by the *equality reflection* rule [4]

$$\frac{\Gamma \vdash e : \mathsf{Eq}_T(e_1, e_2)}{\Gamma \vdash e_1 \equiv e_2 : T}$$

The rule has the additional disadvantage of making judgmental equality undecidable. Nevertheless, this is the type theory we want to implement because it has a great deal of expressive power. The user may essentially adjoin new judgmental equalities by hypothesizing inhabitants of the corresponding equality types, and thus axiomatize many type-theoretic constructions (sums, propositional truncation, (co)inductive types, inductive-inductive and inductive-recursive types, etc.). In contrast to the J-rule of intensional type theory, equality reflection erases uses of equality proofs, which ought to prove useful in certain kinds of formalization. By combining equality reflection with handlers, described below, the user also controls opacity of definitions and application of (user-provided) normalization strategies.

The *AML evaluator* performs bidirectional type checking of terms, invoking operations (questions) that can be handled (answered) by user-provided AML code in the style of Eff [2]. For instance, to construct a well-typed application $e_1\, e_2$, the first step is to synthesize the type T_1 of e_1 and express it as a product. Since type theory with equality reflection does not enjoy strong normalization, the evaluator simply triggers an operation `as_prod(`$\Gamma \vdash T_1$: `Type)` and expects a handler to yield back an inhabitation judgment $\Gamma \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(T_1, \prod_{(x:A)} B)$ for some A, B, and ξ. Once it has the appropriate premises, the evaluator invokes the nucleus to construct the resulting judgment $\Gamma \vdash e_1\, e_2 : B[e_2/x]$ (with some typing annotations elided). There is also an operation `as_eq` for asking how to convert a type to an equality type.

Similarly, whenever the evaluator encounters a non-trivial equality $\Gamma \vdash e_1 \equiv e_2 : T$ it triggers the operation `equal(`$\Gamma \vdash e_1 : T$`)(`$\Gamma \vdash e_2 : T$`)`, and the equation is verified if and when the handler yields a judgment $\Gamma \vdash e : \mathsf{Eq}_T(e_1, e_2)$ back to the evaluator. The handler can employ an arbitrary equality checking algorithm, using support from the nucleus to produce witnesses for the β-rule, function extensionality, η-rules for records, uniqueness of equality proofs, and congruence rules.

```
λ (θ : a ≡ b) (ξ : b ≡ c),
  handle θ : a ≡ c with
  | equal ((⊢ a ≡ b) as ?X) ((⊢ a ≡ c) as ?Y) =>
      handle yield (congruence X Y) with
      | equal (⊢ b) (⊢ c) => yield (Some ξ)
      end
  end
```

Listing 1: Transitivity of equality

In practice most equalities can be verified by a standard type-directed equality checking algorithm. We have implemented such an algorithm in AML and extended it with *equality hints*. These allow the user to dynamically add extensionality rules, β-rules, and instructions on how to immediately resolve equalities that are not amenable to rewriting (such as commutativity of addition). The fact that the algorithm is implemented in AML gives it a strong correctness guarantee.

For example, given a type T with elements $a, b, c : T$, the AML program[1] in Listing 1 computes a witness of $\mathsf{Eq}_T(a,b) \to \mathsf{Eq}_T(b,c) \to \mathsf{Eq}_T(a,c)$, namely the term $\lambda\theta\,\xi\,.\,\theta$ (where type annotations have been elided). While verifying that θ does have type $\mathsf{Eq}_T(a,c)$, the nucleus encounters a non-trivial equality of types $\mathsf{Eq}_T(a,b)$ and $\mathsf{Eq}_T(a,c)$. The outer handler handles this by an application of `congruence`, which attempts to generate a witness of equality by applying congruence rules. Structural comparison of $\mathsf{Eq}_T(a,b)$ and $\mathsf{Eq}_T(a,c)$ generates three further equality checks, of which $T \equiv_{\mathsf{Type}} T$ and $a \equiv_T a$ are trivial, and $b \equiv_T c$ is handled by the hypothesis ξ. In general we do not expect users to write such low-level handlers but rather rely on a sophisticated standard library provided by the developers.

Much work remains to be done. We plan to introduce mechanisms in the AML evaluator that will allow users to implement implicit coercions, type classes, and universes. For a more substantial example we plan to implement Voevodsky's Homotopy Type System [5] as a way of introducing intensional identity types in Andromeda.

References

[1] The Andromeda theorem prover. https://github.com/Andromedans/andromeda/tree/TYPES2016.

[2] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[3] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[4] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

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

[1] In AML, $e_1 \equiv e_2$ can be read as $\mathsf{Eq}_T(e_1, e_2)$, and handlers return their answers using `yield`; see [1] for details.

Extracting a formally verified Subtyping Algorithm for Intersection Types from Ideals and Filters

Jan Bessai[1], Andrej Dudenhefner[1], Boris Düdder[1], and Jakob Rehof[1]

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

Abstract

The BCD type system of intersection types has been introduced by Barendregt, Coppo and Dezani in [1]. It is derived from a filter lambda model in order to characterize exactly the strongly normalizing terms. Formally, intersection types over variables $\alpha \in \mathbb{V}$

$$\sigma, \tau, \rho ::= \alpha \mid \sigma \to \tau \mid \sigma \cap \tau \mid \omega$$

are related by the least preorder $\leq$ closed under the rules

$$\sigma \leq \omega, \quad \omega \leq \omega \to \omega, \quad \sigma \cap \tau \leq \sigma, \quad \sigma \cap \tau \leq \tau, \quad \sigma \leq \sigma \cap \sigma;$$
$$(\sigma \to \tau) \cap (\sigma \to \rho) \leq \sigma \to \tau \cap \rho;$$
$$\text{If } \sigma \leq \sigma' \text{ and } \tau \leq \tau' \text{ then } \sigma \cap \tau \leq \sigma' \cap \tau' \text{ and } \sigma' \to \tau \leq \sigma \to \tau'.$$

Decidability of this preorder has been shown in [6, 4, 7, 8]. Laurent has formalized the relation in Coq in order to eliminate transitivity cuts from it [5]. Following the ideas presented in [8], we show how to obtain a formally verified subtyping algorithm in Coq. Focusing on the algebraic properties of filters and ideals on the subtype relation, we manage to avoid additional proof infrastructure (e.g. lists of types) and extensions to the core type theory of Coq. When executed inside Coq, the algorithm produces a subtype proof tree for an arbitrary pair of intersection types or a counter proof if the input pair is not subtype related.

Automatic program extraction allows to obtain Haskell and OCaml versions of the algorithm. Extracted code can be used as a reference for randomized testing of manually optimized implementations. We will report on an implemented but not yet machine verified subtype algorithm with $\mathcal{O}(n^2)$ asymptotic runtime behavior.

Proven properties allow to formally show the correspondence between prime ideals and the notion of paths in intersection types, which is mentioned in [9]. Organization into an intersection of paths is an important lemma in proofs for various decision problems, e.g. type inhabitation [3], type matching [2] and type inference [4]. We will demonstrate our implementation and made it publicly[1] available in the hope that it can serve as a platform for exploring formal verification and program extraction of algorithms based on intersection types.

References

[1] 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.

[2] Boris Düdder, Moritz Martens, and Jakob Rehof. Intersection Type Matching with Subtyping. In *Proceedings of TLCA'13*, volume 4(6) of *LNCS*. http://dx.doi.org/10.4230/DagRep.4.6.29, 2013.

[1] https://www.github.com/JanBessai/BCD

[3] Boris Düdder, Moritz Martens, Jakob Rehof, and Paweł Urzyczyn. Bounded Combinatory Logic. In *Proceedings of CSL'12*, volume 16 of *LIPIcs*, pages 243–258. Schloss Dagstuhl, 2012.

[4] T. Kurata and M. Takahashi. Decidable properties of intersection type systems. In *TLCA*, volume 902 of *LNCS*, pages 297–311. Springer, 1995.

[5] Olivier Laurent. Intersection types with subtyping by means of cut elimination. *Fundamenta Informaticae*, 121(1-4):203–226, 2012.

[6] Benjamin C Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical Report CMU-CS-89-1693, CMU, 1989.

[7] Jakob Rehof and Paweł Urzyczyn. Finite Combinatory Logic with Intersection Types. In *Proceedings of TLCA'11*, volume 6690 of *LNCS*, pages 169–183. Springer, 2011.

[8] Rick Statman. A Finite Model Property for Intersection Types. In Jakob Rehof, editor, *Proceedings Seventh Workshop on Intersection Types and Related Systems, ITRS 2014, Vienna, Austria, 18 July 2014.*, volume 177 of *EPTCS*, pages 1–9, 2015.

[9] Steffen Van Bakel, Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Fer-Jan de Vries. Intersection types for λ-trees. *Theoretical Computer Science*, 272(1):3–40, 2002.

Rank 3 Inhabitation of Intersection Types Revisited

Jan Bessai[1], Andrej Dudenhefner[1], Boris Düdder[1], and Jakob Rehof[1]

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

Abstract

Intersection types capture deep semantic properties of λ-terms such as normalization and were subject to extensive study for decades [2]. Due to their expressive power, intersection type inhabitation (given a type, does there exist a term having the type?) is undecidable in the standard intersection type system **BCD** [1].

Urzyczyn showed relatively recently [6] that restricted to rank 2 intersection type inhabitation becomes exponential space complete and is undecidable for rank 3 and up. Here, $\mathrm{rank}(\tau) = 0$ if τ is a simple type, $\mathrm{rank}(\sigma \to \tau) = \max(\mathrm{rank}(\sigma) + 1, \mathrm{rank}(\tau))$ and $\mathrm{rank}(\sigma \cap \tau) = \max(1, \mathrm{rank}(\sigma), \mathrm{rank}(\tau))$ otherwise. Later, Salvati et al. discovered the equivalence of intersection type inhabitation and λ-definability [5], which further improved our understanding of the expressiveness of intersection types.

One can take several routes in order to track the reason for undecidability of intersection type inhabitation (abbreviated by IHP, resp. IHP3 for rank 3). In [2] the following reduction is performed: EQA $\leq$ ETW $\leq$ WTG $\leq$ IHP, where EQA is the emptiness problem for queue automata, ETW is the emptiness problem for typewriter automata and WTG is the problem of determining whether one can win a tree game. A different route taken in [6] performs the following reduction: ELBA $\leq$ SSTS1 $\leq$ HETM $\leq$ IHP3, where ELBA is the emptiness problem for linear bounded automata, SSTS1 is the problem of deciding whether there is a word that can be rewritten to 1s in a simple semi-Thue system and HETM is the halting problem for expanding tape machines. Alternatively, following the route of [5] via semantics, the following reduction can be performed: WSTS $\leq$ LDF $\leq$ IHP3, where WSTS is the word problem in semi-Thue systems and LDF is the λ-definability problem. Each of these routes introduces its own machinery that may distract from the initial question. As a result, it is challenging to even give examples of particularly hard inhabitation problem instances, or distinguish necessary properties on a finer scale than the rank restriction. We pinpoint the reason for undecidability by performing a direct reduction from the halting problem for Turing machines to intersection type inhabitation. In particular, we show how arbitrary Turing machine computations can be directly simulated using proof search. The main benefit of the presented approach is its accessibility and simplicity. It can be used to inspect properties of intersection type inhabitation or related systems on a more fine-grained scale than that of rank. Additionally, simulating Turing machine computations by proof search may provide new insights into the line of work, where proof search is regarded as the execution of a logic programming language used for code synthesis [4]. In order to simulate a Turing machine computation, we construct an intersection type $\tau_\star$ that exactly captures individual properties of a Turing machine. Due to the structure of $\tau_\star$, the execution of a proof search algorithm [3] necessarily consists of two phases. During the first phase the simultaneous set of type judgments (used by the algorithm) is expanded to provide room for the simulation of a Turing machine computation. During the second phase the simultaneous set of type judgments is transformed according to the transition function of the Turing machine until the final state is reached. As a result, $\tau_\star$ is inhabitable iff the simulated Turing machine halts. Since our construction is more concise than existing approaches taking no detours, we believe that it is valuable for a better understanding of the expressiveness of intersection type inhabitation.

For the sake of completeness, we outline the construction of $\tau_\star$. Let $M = (\Sigma, Q, q_0, q_f, \delta)$ be a TM. Fix the set of type constants $\mathbb{C} = \Sigma \dot\cup \{l, r, \bullet\} \dot\cup \{\langle q, a\rangle \mid q \in Q, a \in \Sigma\} \dot\cup \{\circ, *, \#, \$\}$. Note that $\llcorner\!\lrcorner \in \Sigma$ is the space symbol. We define the following types:

$$\sigma_f = \bigcap_{c \in \Sigma} (c \cap \langle q_f, c\rangle)$$

$$\text{for each } t = ((q, c) \mapsto (q', c', +1)) \in \delta$$

$$\sigma_t = \bigcap_{a \in \Sigma} (\bullet \to a \to a) \cap (l \to c' \to \langle q, c\rangle) \cap \bigcap_{a \in \Sigma} (r \to \langle q', a\rangle \to a)$$

$$\text{for each } t = ((q, c) \mapsto (q', c', -1)) \in \delta$$

$$\sigma_t = \bigcap_{a \in \Sigma} (\bullet \to a \to a) \cap (r \to c' \to \langle q, c\rangle) \cap \bigcap_{a \in \Sigma} (l \to \langle q', a\rangle \to a)$$

$$\sigma_* = ((\bullet \to \circ) \to \circ) \cap ((\bullet \to *) \to *) \cap ((l \to *) \to \#) \cap ((r \to \#) \cap (\bullet \to \$) \to \$)$$

$$\sigma_0 = ((\bullet \to \langle q_0, \llcorner\!\lrcorner\rangle) \to \circ) \cap ((\bullet \to \llcorner\!\lrcorner) \to *) \cap ((l \to \llcorner\!\lrcorner) \to \#) \cap ((r \to \llcorner\!\lrcorner) \to \$)$$

$$\tau_\star = \sigma_0 \to \sigma_* \to \sigma_f \to \sigma_{t_1} \to \ldots \to \sigma_{t_k} \to (l \to \circ) \cap (r \to \#) \cap (\bullet \to \$)$$

$$\text{where } \delta = \{t_1, \ldots, t_k\}$$

Each of the above types represents one particular feature of M. To provide some intuition:

- l and r link neighboring tape cells.
- $\circ$, \$ and # indicate the first, last and next to the last tape cell during tape expansion.
- $\langle q, a\rangle$ represents that M is at the position reading a in state q.
- σ_f recognizes whether an accepting state is reached.
- σ_t transforms the state according to δ.
- σ_* represents tape expansion.
- σ_0 initializes M to the state q_0 and the empty tape.
- $\tau_\star$ is inhabited iff M accepts the empty word.

References

[1] 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.

[2] H.P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic, Cambridge University Press, 2013.

[3] Martin W. Bunder. The inhabitation problem for intersection types. In James Harland and Prabhu Manyem, editors, *Theory of Computing 2008. Proc. Fourteenth Computing: The Australasian Theory Symposium (CATS 2008), Wollongong, NSW, Australia, January 22-25, 2008. Proceedings*, volume 77 of *CRPIT*, pages 7–14. Australian Computer Society, 2008.

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

[5] S. Salvati, G. Manzonetto, M. Gehrke, and H.P. Barendregt. Urzyczyn and Loader are logically related. In *Proceedings of ICALP 2012*, volume 7392 of *LNCS*, pages 364–376. Springer, 2012.

[6] P. Urzyczyn. Inhabitation of Low-Rank Intersection Types. In *Proceedings of TLCA'09*, volume 5608 of *LNCS*, pages 356–370. Springer, 2009.

Guarded cubical type theory

Lars Birkedal[1], Aleš Bizjak[1], Ranald Clouston[1], Hans Bugge Grathwohl[1], Bas Spitters[1], and Andrea Vezzosi[2]

[1] Department of Computer Science, Aarhus University
[2] Department of Computer Science and Engineering, Chalmers University of Technology

Guarded dependent type theory [1] is a dependent type theory with guarded recursive types, which are useful for building models of program logics, and as a tool for programming and reasoning with coinductive types. This is done via a modality $\triangleright$, pronounced 'later', with a constructor next, and a *guarded* fixed-point combinator $\mathsf{fix} : (\triangleright A \to A) \to A$. This combinator is used both to define guarded recursive types as fixed-points of functions on universes, as well as to define functions on these types.

Cubical type theory [2] is an extension of Martin-Löf type theory with the goal of obtaining a computational interpretation of the univalence axiom. The important novel idea in cubical type theory is that the identity type is not inductively defined. Instead the identity type[1] $\mathsf{Id}_A(x, y)$ is defined to be the type of *paths* starting from x to y. These paths are defined via an abstract interval $\mathbb{I}$ with endpoints $0, 1$. Elements of the identity type are then introduced using path abstraction: if t is a term of type A in context $\Gamma, i : \mathbb{I}$, then $\langle i \rangle\, t$ is a term of type $\mathsf{Id}_A(t[0/i], t[1/i])$ in context Γ. The elimination rule for the identity type is application: given an element i of $\mathbb{I}$ and a proof $p : \mathsf{Id}_A(x, y)$ the term $p\, i$ is of type A, with two judgemental equalities: $p\, 0 \equiv x$ and $p\, 1 \equiv y$. Several extensionality properties are derivable in this type theory. In particular, function extensionality is provable with the term

$$\mathsf{funext} = \lambda f g p.\, \langle i \rangle\, \lambda a.(p a)\, i : \prod_{f g : A \to B} \prod_{x : A} (\mathsf{Id}_B(f x, g x) \to \mathsf{Id}_{A \to B}(f, g)). \tag{1}$$

It is hoped that this type theory has decidable type-checking and satisfies canonicity.

We propose *guarded cubical type theory*, a combination of the two type theories with the goal of obtaining a type theory with a later modality and guarded fixed-point combinator that has decidable type-checking and satisfies canonicity. In previous work on guarded dependent type theory the focus was on designing the rules of the type theory so that it is possible to work with guarded recursive types. In particular, the type theory in [1] is an extensional type theory, i.e., there is the equality reflection rule for the identity type, and there is the judgemental equality

$$\mathsf{fix}\, f \equiv f(\mathsf{next}(\mathsf{fix}\, f)). \tag{2}$$

Both of these prevent decidable type-checking. Moreover, the type theory in [1] also includes

$$\mathsf{com} : \triangleright\, \mathsf{Id}_A(x, y) \to \mathsf{Id}_{\triangleright A}(\mathsf{next}\, x, \mathsf{next}\, y)$$

which is the inverse to the canonical term of type $\mathsf{Id}_{\triangleright A}(\mathsf{next}\, x, \mathsf{next}\, y) \to \triangleright\, \mathsf{Id}_A(x, y)$. We found that com is crucial for proving properties of guarded recursive types. Such a term com is not definable in Martin-Löf type theory when the identity type is defined in the usual way and so we need to introduce it axiomatically, which leads to the loss of canonicity. Note the formal resemblance of the type of com to the type of the axiom of function extensionality in (1).

[1] We do not distinguish between the types Id and Path in the interest of presentation.

Guarded cubical type theory Using the constructs from cubical type theory we can address both of these difficulties. First, because the identity type has a much more flexible introduction rule, the term com becomes definable as

$$\mathsf{com} = \lambda\,(p : \triangleright\,\mathsf{Id}_A\,(x, y))\,.\,\langle i\rangle\,\mathsf{next}\,[p' \leftarrow p]\,.p'i,$$

which should be compared to the term funext in (1). Here $[p' \leftarrow p]$ is a *delayed substitution*. Delayed substitutions, introduced in [1], are a generalisation of the applicative functor structure of $\triangleright$ to dependent types.

Interestingly, and rather surprisingly, we can utilise the *face lattice* $\mathbb{F}$ of cubical type theory to control fixed-point unfolding. We omit the fixed-point unfolding rule (2) from the type theory and instead decorate the fixed-point with a face which specifies when the fixed-point should be unfolded. The typing rule for fix becomes (omitting some details for this abstract):

$$\frac{\Gamma \vdash t : \triangleright A \to A \qquad \Gamma \vdash \phi : \mathbb{F}}{\Gamma \vdash \mathsf{fix}^{\phi}\,t}$$

with the only judgemental equality rule being $\mathsf{fix}^{1_{\mathbb{F}}}\,t \equiv f\,\left(\mathsf{next}\,\left(\mathsf{fix}^{0_{\mathbb{F}}}\,t\right)\right)$, where $1_{\mathbb{F}}$ and $0_{\mathbb{F}}$ are the top and bottom elements of $\mathbb{F}$. This judgemental equality rule allows us to unfold the fixed-point on demand. The face ϕ associated to the fixed-point can then be used to prove

$$\langle i\rangle\,\mathsf{fix}^{i=1}\,f : \mathsf{Id}_A\,\left(\mathsf{fix}^{0_{\mathbb{F}}}\,f, f\,\left(\mathsf{next}\,\left(\mathsf{fix}^{0_{\mathbb{F}}}\,f\right)\right)\right).$$

Prototype implementation Our prototype implementation of the type theory[2] extends the cubical type checker[3] with $\triangleright$ modality and guarded fixed-points. One of the motivations behind cubical type theory is to have some extensionality, while preserving the strong metatheoretic properties of intensional type theory. Our experiments with the prototype implementation show that the examples described in [1] can indeed be expressed in guarded cubical type theory, albeit with some more manual rewriting, since fixed-point unfolding is no longer a judgemental equality, but only a propositional one. This is a confirmation that the cubical type theory is relevant not only for mathematical applications, but also in areas of computer science.

Semantics We are working on a semantics of guarded cubical type theory based on an axiomatic version of the cubical model in the internal logic of cubical sets [3, 4]. Our axioms include a presheaf topos with an internal De Morgan algebra having the disjunction property and an internal operator $\forall$. The verification that these axioms actually suffice is nearly done.

The concrete model we use is based on presheaves over the product of the category of cubes (as used in the model of cubical type theory) and the preorder ω (as used in the model of guarded dependent type theory), and this presheaf topos satisfies our axioms.

References

[1] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In *FoSSaCS 2016*, 2016.

[2] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. Unpublished, 2016.

[3] Thierry Coquand. Internal version of the uniform Kan filling condition. Unpublished, 2015.

[4] Bas Spitters. Cubical sets as a classifying topos. *TYPES'15*, 2015.

[2] https://github.com/hansbugge/cubicaltt/tree/gcubical

[3] https://github.com/mortberg/cubicaltt

Hybrid realizability for intuitionistic and classical choice*

Valentin Blot

University of Bath

Realizability appeared in [Kle45] as a formal account of the Brouwer-Heyting-Kolmogorov interpretation of logic, leading to the Curry-Howard isomorphism between intuitionistic proofs and purely functional programs. In [Gri90], Griffin used control operators to extend the isomorphism to classical logic. While the first realizability interpretations of classical logic relied on a negative translation followed by an intuitionistic realizability interpretation, Griffin's discovery can be exploited to give a direct realizability interpretation of classical logic. This allowed Krivine to interpret second-order Peano arithmetic and the axiom of dependent choice in an untyped λ-calculus extended with the `call/cc` operator [Kri09]. In the work presented here, we interpret first-order classical arithmetic and the axiom of countable choice in a model of the simply-typed $\lambda\mu$-calculus [Par92], an extension of λ-calculus with control features.

The axiom of choice is ubiquitous in mathematics and is often used without even noticing. Therefore, having a computational interpretation of this axiom is essential to the extraction of programs from a wide range of mathematical proofs. While in usual interpretations of intuitionistic logic the general axiom of choice is trivially realized, interpreting even its countable version in a classical setting requires the use of strong recursion schemes, like the bar recursion operator [Spe62]. The requirement for such a strong recursion principle can be explained by the much stronger provability strength of classical choice: since any formula can be reflected by a boolean in classical logic, the axiom of countable choice can build the characteristic function of any formula with integer parameters, even the undecidable ones.

Variants of bar recursion were used in [BBC98, BO05] to interpret the negative translation of the axiom of choice in an intuitionistic setting. In [BR13], it was shown that bar recursion can be used in a language with control operators to interpret directly the axiom of countable choice in a classical setting. In the present work we extend this approach, adding strong existentials to the realizability interpretation. Because classical proofs typically involve some backtracking, strong existentials are problematic in a classical setting [Her05]. The reason is that the witness of a strong existential may change when the exploration of the associated proof performs some backtrack. Conversely, weak existentials (which, in our setting, are double negations of the strong ones) work well with control operators, but are less efficient from the computational perspective because their interpretations indeed involve backtracking.

Rather than having to choose between an efficient system which is restricted to intuitionistic logic or a full classical system with a complex computational interpretation, we take the best from both worlds and work within classical logic with strong existential quantifications, using their weak counterparts when classical reasoning is needed. In a proof where the excluded middle is never used on some existential formula, this existential can be strong and benefit from an efficient interpretation (in particular, the general axiom of choice is trivially realized in that case). If in the same proof some classical reasoning is performed on another existential formula, then that existential must be weak, and while we can still use the axiom of countable choice on it, its computational interpretation is given by bar recursion and can involve a costly recursion. Restricting strong existentials to intuitionistic logic relies on the polarities of [Blo15] which forbids classical reasoning on strong existentials and ensures the correctness of our computational interpretation in $\lambda\mu$-calculus. Similarity between our polarities and those of other

*Research supported by the UK EPSRC grant EP/K037633/1.

proof systems like Girard's LU [Gir93] and LC [Gir91], or Liang and Miller's PCL [LM13] are still to be investigated. Combination of strong existentials and classical logic was also investigated in [Her12], where strong existentials were weakened enough to make them compatible with classical logic while preserving countable and dependent choice. We believe that there are strong connections between the operational semantics of Herbelin's calculus and bar recursion.

In our setting, witnesses can be extracted from proofs of strong existentials, as well as from proofs of Π^0_2 formulas with a weak existential. This second case relies on a standard non-empty realizability interpretation of the false formula. Our system allows for extraction of more efficient programs than with the usual direct or indirect interpretations of classical logic, provided some care is taken to choose strong existentials whenever possible.

This work will be presented at the LICS 2016 conference in July.

References

[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.

[Blo15] Valentin Blot. Typed realizability for first-order classical analysis. *Logical Methods in Computer Science*, 11(4), 2015.

[BO05] Ulrich Berger and Paulo Oliva. Modified bar recursion and classical dependent choice. In *Logic Colloquium '01, Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic*, volume 20 of *Lecture Notes in Logic*, pages 89–107. A K Peters, Ltd., 2005.

[BR13] Valentin Blot and Colin Riba. On Bar Recursion and Choice in a Classical Setting. In *11th Asian Symposium on Programming Languages and Systems*, volume 8301 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2013.

[Gir91] Jean-Yves Girard. A New Constructive Logic: Classical Logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.

[Gir93] Jean-Yves Girard. On the Unity of Logic. *Annals of Pure and Applied Logic*, 59(3):201–217, 1993.

[Gri90] Timothy Griffin. A Formulae-as-Types Notion of Control. In *17th Symposium on Principles of Programming Languages*, pages 47–58. ACM Press, 1990.

[Her05] Hugo Herbelin. On the Degeneracy of Sigma-Types in Presence of Computational Classical Logic. In *7th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Mathematics, pages 209–220. Springer, 2005.

[Her12] Hugo Herbelin. A Constructive Proof of Dependent Choice, Compatible with Classical Logic. In *27th IEEE Symposium on Logic in Computer Science*, pages 365–374. IEEE Computer Society, 2012.

[Kle45] Stephen Cole Kleene. On the Interpretation of Intuitionistic Number Theory. *Journal of Symbolic Logic*, 10(4):109–124, 1945.

[Kri09] Jean-Louis Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009.

[LM13] Chuck Liang and Dale Miller. Unifying Classical and Intuitionistic Logics for Computational Control. In *28th ACM/IEEE Symposium on Logic in Computer Science*, pages 283–292. IEEE Computer Society, 2013.

[Par92] Michel Parigot. $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *3rd International Conference on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

[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.

Parametricity and excluded middle

Auke Booij

University of Birmingham

Abstract

In univalent foundations, it is known that the law of excluded middle allows one to define a family of functions $f_X : X \to X$ that is not the identity function on the booleans. We show that the converse holds as well: given such a function, we derive the law of excluded middle.

Suppose we are given a polymorphic function

$$f_X : X \to X,$$

where $X : \mathcal{U}$ is its type parameter.

If this were a term in a language such as System F, then parametricity tells us that it must be equal to the identity function id_X for every type X. But parametricity is a metatheoretical framework: it gives properties about the terms of a language, rather than internally stating properties of elements.

Internal to univalent foundations, if we have LEM, then there exists a polymorphic function f such that $f_{\mathbf{2}}$ (where $\mathbf{2}$ is the type of booleans) is not the identity function [2, exercise 6.9]. Since LEM is consistent with univalent foundations, this means that there cannot be an internal proof that a polymorphic function $f_X : X \to X$ is equal to the identity.

We prove that, in univalent foundations, LEM is precisely what is needed to get a function family not equal to the identity on $\mathbf{2}$: on the one hand, we already know that LEM gives us such a function; on the other hand, we have the following converse.

Theorem 1. *If there is a function $f : \Pi_{X:\mathcal{U}} X \to X$ with $f_{\mathbf{2}} \neq \mathsf{id}_{\mathbf{2}}$, then LEM holds.*

Alternatively, to confine the amount of univalence needed, we can work in the setting of intensional type theory with function extensionality (but without full univalence), and assume that f is *extensional* in the sense that it is invariant under equivalences on the type X it acts on.

The idea of the proof is that we define a type $\mathbf{3}_P$, which, depending on whether P holds, may or may not be equivalent to $\mathbf{2}$. We then evaluate f at the type $\mathbf{3}_P \simeq \mathbf{3}_P$ (rather than $\mathbf{3}_P$ itself), and prove $P + \neg P$ using that evaluation.

This proof has been formalized [1] in Agda using the HoTT library.

Proof. Without loss of generality, we may assume that $f_{\mathbf{2}}(0_{\mathbf{2}}) \neq 0_{\mathbf{2}}$.

To prove LEM, let P be an arbitrary proposition. We need to prove $P + \neg P$.

We will consider a type with three points, where we identify two points depending on whether P holds. Formally, this is the quotient of a three-element type, where the relation between two of those points is the proposition P. This quotient can be constructed conveniently as

$$\mathbf{3}_P := \Sigma P + \mathbf{1},$$

where ΣP is the *suspension* of P[1]. The two points of the suspension are called N and S, and the identity path (if it exists) between those points is called $\mathsf{merid}(p) : \mathsf{N} = \mathsf{S}$, with $p : P$.

Recall the following about suspensions.

[1] The suspension of a type is not generally a quotient, because it is not generally a set: we use the fact that P is a proposition here.

- By induction, we can define a map
$$\mathsf{swap} : \Sigma P \to \Sigma P$$
that sends N to S and vice versa.
- By induction, we can define a map $\mathsf{extract} : \mathsf{N} =_{\Sigma P} \mathsf{S} \to P$, and this can be generalized to a map
$$\mathsf{extract}'_x : x =_{\Sigma P} \mathsf{swap}(x) \to P.$$

Notice that if we have P, then the suspension is contractible, so $\mathbf{3}_P \simeq \mathbf{2}$, and also $(\mathbf{3}_P \simeq \mathbf{3}_P) \simeq \mathbf{2}$.

Define
$$g := f_{\mathbf{3}_P \simeq \mathbf{3}_P}(\mathsf{ide}_{\mathbf{3}_P}) : \mathbf{3}_P \simeq \mathbf{3}_P,$$
where $\mathsf{ide}_{\mathbf{3}_P}$ is the equivalence $\mathbf{3}_P \simeq \mathbf{3}_P$ given by the identity function on $\mathbf{3}_P$. We will see g both as an equivalence and as a function $\mathbf{3}_P \to \mathbf{3}_P$.

Now we do case analysis on $g(\mathsf{inr}(\star))$. Notice that this case analysis is simply an instance of the induction principle for sum types. In particular, we do not require decidable equality of $\mathbf{3}_P$ (which would already give us $P + \neg P$, which is exactly what we are trying to prove). When analyzing the case $\mathsf{inr}(t) : \mathbf{3}_P$, with $t : \mathbf{1}$, we are free to specialize to $t = \star$ since $\mathbf{1}$ is contractible.

$g(\mathsf{inr}(\star)) = \mathsf{inr}(\star)$**:** Assume that P holds. Then by transporting the witness of $f_{\mathbf{2}}(0_{\mathbf{2}}) \neq 0_{\mathbf{2}}$ along an equivalence that identifies $0_{\mathbf{2}}$ with $\mathsf{ide}_{\mathbf{3}_P}$, we get that $g \neq \mathsf{ide}_{\mathbf{3}_P}$. However, since $\mathbf{3}_P \simeq \mathbf{2}$ and g has a fixed point $\mathsf{inr}(\star)$, we can deduce that $g = \mathsf{ide}_{\mathbf{3}_P}$, which is a contradiction.

$g(\mathsf{inr}(\star)) = \mathsf{inl}(x)$**:** We do further case analysis on $g(\mathsf{inl}(x))$.

$g(\mathsf{inl}(x)) = \mathsf{inr}(\star)$**:** We do further case analysis on $g(\mathsf{inl}(\mathsf{swap}(x)))$.

$g(\mathsf{inl}(\mathsf{swap}(x))) = \mathsf{inr}(\star)$**:** Since we now have
$$g(\mathsf{inl}(x)) = \mathsf{inr}(\star) = g(\mathsf{inl}(\mathsf{swap}(x)))$$
and since g is an equivalence, we can use $\mathsf{extract}'_x$ to get P.

$g(\mathsf{inl}(\mathsf{swap}(x))) = \mathsf{inl}(y)$**:** Assume P, in which case $x = \mathsf{swap}(x)$. Hence $\mathsf{inr}(\star) = \mathsf{inl}(y)$ which is a contradiction.

$g(\mathsf{inl}(x)) = \mathsf{inl}(y)$**:** Assume P, in which case $\mathsf{inl}(x) = \mathsf{inl}(y)$. But we now have
$$g(\mathsf{inr}(\star)) = \mathsf{inl}(x) = \mathsf{inl}(y) = g(\mathsf{inl}(x)).$$
So since g is an equivalence, this yields $\mathsf{inr}(\star) = \mathsf{inl}(x)$, which is a contradiction.

□

Acknowledgements

Thanks to Martín Escardó, my supervisor, for his support. Thanks to Uday Reddy for giving the talk on parametricity that inspired me to think about this.

References

[1] A. B. Booij. *Parametricity and excluded middle in Agda*. University of Birmingham, UK. URL: http://www.cs.bham.ac.uk/~abb538/agda/nonparametric.html.

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

The quaternionic Hopf fibration in homotopy type theory

Ulrik Buchholtz and Egbert Rijke

Carnegie Mellon University, Pittsburgh, USA
{ulrikb,erijke}@andrew.cmu.edu

The book on homotopy type theory [5] presents a beginning of the field of synthetic homotopy theory, which carries out homotopy-theoretical constructions internally in type theory. This makes it possible to very directly reason about abstract homotopy theory using proof assistants for dependent type theories, and as such construes homotopy theory as a branch of logic.

One high point in the book is the construction of the Hopf fibration $S^1 \hookrightarrow S^3 \to S^2$. Classically, this is obtained by the so-called Hopf construction from the multiplication of unit-length complex numbers (that is, the unit circle in the complex plane), but in homotopy type theory we can reason about the circle in another way, namely as a higher inductive type, generated by a point constructor base and a path constructor $\mathsf{loop} : \mathsf{base} = \mathsf{base}$.

The Hopf construction applies to a connected H-space A, that is, a path-connected type A together with a binary operation μ and an element e that is neutral for this operation up to homotopy: $\mu(x, e) = x = \mu(e, x)$ for $x : A$ (the equality symbol refers to the identity type). The result is a fibration $\mathsf{hopf} : \Sigma A \to \mathsf{Type}$, where ΣA denotes the suspension of A. The total space can then be identified with the join $A * A$ of A with itself (the join $A * B$ of two types A and B is defined as the pushout of the two projections from $A \times B$, taking advantage of the fact that a pushout in type theory denotes a homotopy pushout of the corresponding spaces). Note that the join of two spheres is again a sphere: $S^n * S^m \simeq S^{n+m+1}$.

The Hopf fibration is in fact but one of a family of four fundamental fibrations in homotopy theory built from the classical normed division algebras $\mathbb{R}$, $\mathbb{C}$, $\mathbb{H}$, $\mathbb{O}$ (the real numbers, the complex numbers, the quaternions and the octonions). From these we classically obtain H-space structures on the unit spheres, S^0, S^1, S^3, S^7, inside these algebras, and from these the corresponding Hopf fibrations:

$$S^0 \hookrightarrow S^1 \to S^1$$
$$S^1 \hookrightarrow S^3 \to S^2$$
$$S^3 \hookrightarrow S^7 \to S^4$$
$$S^7 \hookrightarrow S^{15} \to S^8$$

Coming back to HoTT, it has been an open problem to construct the quaternionic Hopf fibration corresponding to the multiplication of unit quaternions (as $\mathbb{H}$ is a 4-dimensional algebra, these form a three-sphere, S^3).

Here we present a solution to this problem based on a modification of the Cayley-Dickson construction of the normed division algebras $\mathbb{R}$, $\mathbb{C}$, $\mathbb{H}$, $\mathbb{O}$.

The Cayley-Dickson construction (as described for instance by Baez [1]) produces a new $*$-algebra $A' := A \oplus A$ from a given one A by the stipulations

$$(a, b)^* := (a^*, -b), \quad (a, b)(c, d) := (ac - db^*, a^*d + cb).$$

This procedure cannot be productively replicated in homotopy type theory for our purpose, as any vector space over the reals is contractible as a homotopy type.

We were able to find an analog in the setting of synthetic homotopy theory by concentrating on the *unit imaginaries* inside the classical $*$-algebras, and we give a construction that produces

an H-space structure on $\Sigma A * \Sigma A$ for any type A with an involutive negation for which the suspension ΣA has an associative H-space structure that interacts nicely with the negation on A.

Applying the construction to the 0-sphere (and thus to the multiplication on its suspension, the circle), we obtain an H-space structure on the three-sphere, and thus the quaternionic Hopf fibration $S^3 \hookrightarrow S^7 \to S^4$. A famous application of the quaternionic Hopf fibration is as one ingredient in Milnor's construction of exotic 7-spheres [3], that is, smooth manifold structures on the S^7 that are not diffeomorphic to the standard smooth structure. This application, however, is not yet within reach of synthetic homotopy theory.

Our results have been formalized in the Lean proof assistant [4]. Lean has built-in support for HoTT in that it provides two kinds of higher inductive types, namely type quotients and n-truncations, from which a whole host of other common HITs can be deduced, including pushouts, joins, spheres, etc.

The formalization makes heave use of the cubical methods developed in [2], which leverage indexed inductive types representing squares and cubes in types, and paths, squares and cubes lying over given paths, squares and cubes in type families.

References

[1] John C. Baez. The octonions. *Bull. Amer. Math. Soc. (N.S.)*, 39(2):145–205, 2002.

[2] Daniel R. Licata and Guillaume Brunerie. A cubical approach to synthetic homotopy theory. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, LICS '15, pages 92–103, Washington, DC, USA, 2015. IEEE Computer Society.

[3] John Milnor. On manifolds homeomorphic to the 7-sphere. *Ann. of Math. (2)*, 64:399–405, 1956.

[4] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *Automated Deduction – CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388. Springer International Publishing, Cham, 2015.

[5] The Univalent Foundations Program. *Homotopy type theory—univalent foundations of mathematics.* The Univalent Foundations Program, Princeton, NJ; Institute for Advanced Study (IAS), Princeton, NJ, 2013.

Towards a Logic of Multi-Party Sessions

Marco Carbone
IT University of Copenhagen
Copenhagen
Denmark
maca@itu.dk

Fabrizio Montesi
University of Southern Denmark
Odense
Denmark
famontesi@gmail.com

Carsten Schürmann
IT University of Copenhagen
Copenhagen
Denmark
carsten@itu.dk

Nobuko Yoshida
Imperial College
London
United Kingdom
n.yoshida@imperial.ac.uk

A two-party computation corresponds to the interaction of two processes and a multi-party computation corresponds therefore to the interaction of multiple (three or more) processes. The process algebra community has developed a thorough understanding of multi-party computation through a mechanism called *global type* that determines a sequential order of the individual send and receive actions among the participating parties.

Viewed from the vantage point of logic, we observe that two-party computations are also well understood through a Curry-Howard correspondence that was first pointed out by Caires and Pfenning in the setting of intuitionistic logic [CP10] and then by Wadler in the setting of classical logic [Wad14]. In the binary case, two processes form a two-party computation if their respective types are dual to each other. Inspired by this, we show how to generalize the notion of duality to coherence, which allows us to capture the essence of multi-party computations and establish a Curry-Howard correspondence between multi-party sessions and an extension of linear logic with coherence proofs. This paper is based on prior work in [CMSY15].

In the reminder of the paper, we illustrate the main contributions of this work using the classical 2-buyer protocol [HYC08] as example. Two buyers B1 and B2 attempt to buy a book together from seller S. B1 sends the title of the book that he intends to purchase. The seller S replies to both, B1 and B2, with a quote. B1 then sends a message to B2 about how much money he is willing to contribute to the purchase, leaving B2 with the decision either to contribute the remaining funds and to complete the purchase or to not buy the book at all. Formally, we write:

$$\begin{array}{ll} 1. & \mathsf{B1} \to \mathsf{S} : \langle \mathbf{str} \rangle;\ \mathsf{S} \to \mathsf{B1} : \langle \mathbf{int} \rangle;\ \mathsf{S} \to \mathsf{B2} : \langle \mathbf{int} \rangle;\ \mathsf{B1} \to \mathsf{B2} : \langle \mathbf{int} \rangle; \\ 2. & \mathsf{B2} \to \mathsf{S} : \&\,(\,\mathsf{B2} \to \mathsf{S} : \langle \mathbf{addr} \rangle; \mathsf{end},\ \mathsf{end}) \end{array} \tag{1}$$

Implicitly, for the purpose of this example, we assume that all communication proceeds through a single channel. Its type depends on the role of each party. When one sends, another receives. The following are the types of the shared channels for each role expressed in Wadler's CP. To be consistent with [Wad14] and in a slight deviation from [CDCYP15], we use $\otimes$ to type outputs and $\parr$ to type inputs.

$$\begin{array}{rl} \mathsf{B1}: & \mathbf{str} \otimes \mathbf{int} \parr \mathbf{int} \otimes \mathsf{end} \\ \mathsf{B2}: & \mathbf{int} \parr \mathbf{int} \parr \big((\mathbf{addr} \otimes \mathsf{end}) \oplus \mathsf{end}\big) \\ \mathsf{S}: & \mathbf{str} \parr \mathbf{int} \otimes \mathbf{int} \otimes \big((\mathbf{addr} \parr \mathsf{end}) \,\&\, \mathsf{end}\big) \end{array} \tag{2}$$

Above, each formula in classical linear logic (CLL) states how x is used by each process. For instance, B1 outputs ($\otimes$) a string, receives ($\parr$) an integer, sends another integer and eventually terminates (end).

The motivating observation of this work is that CLL is not general enough to express the composition of three or more processes sharing one channel, since the cut-rule can only compose two processes P and Q on one single shared channel x with a compatible types and A and $A^{\perp}$ and not on three.

$$\frac{P \vdash \Delta, x{:}A \qquad Q \vdash \Delta', x{:}A^{\perp}}{(\nu x{:}A)\,(P \mid Q) \vdash \Delta, \Delta'}\ \mathsf{Cut}$$

As a solution to this challenge, we propose to annotate the connectives with roles as the partner of the communication. For details, consult [CDCYP15].

$$\begin{array}{rl} \mathsf{B1}: & \mathbf{str} \otimes^{\mathsf{S}} \mathbf{int} \parr^{\mathsf{S}} \mathbf{int} \otimes^{\mathsf{B2}} \mathsf{end} \\ \mathsf{B2}: & \mathbf{int} \parr^{\mathsf{S}} \mathbf{int} \parr^{\mathsf{B1}} \left((\mathbf{addr} \otimes^{\mathsf{S}} \mathsf{end}) \oplus^{\mathsf{S}} \mathsf{end}\right) \\ \mathsf{S}: & \mathbf{str} \parr^{\mathsf{B1}} \mathbf{int} \otimes^{\mathsf{B1}} \mathbf{int} \otimes^{\mathsf{B2}} \left((\mathbf{addr} \parr^{\mathsf{B2}} \mathsf{end}) \,\&^{\mathsf{B2}}\, \mathsf{end}\right) \end{array} \tag{3}$$

Annotations identify the dual role of each action, e.g., the usage for B1 now reads: send a string to S ($\otimes^{\mathsf{S}}$); receive an integer from S ($\parr^{\mathsf{S}}$); send an integer to B2 ($\otimes^{\mathsf{B2}}$); and, terminate (end). This trick allows us to generalize the standard notion of de Morgan duality that is defined between two processes of type A and $A^{\perp}$, to coherence between a set of processes $\{A_i\}_i$. Coherence is expressed using the judgment $G \vDash p_1 : A_1, \ldots, p_n : A_n$. Here, G is the global type, each p_i denotes a role of type A_i. Coherence is defined by the following rules.

$$\frac{G \vDash p:A,\ q:C \quad G' \vDash \Theta,\ p:B,\ q:D}{p \to q : \langle G \rangle ; G' \vDash \Theta, p:A \otimes^{q} B,\ q:C \parr^{p} D} \parr \otimes \qquad \frac{}{\mathsf{end} \vDash p:1, q_1:\perp, \ldots, q_n:\perp} \, 1\perp$$

$$\frac{G_1 \vDash \Theta, p:A, q:C \quad G_2 \vDash \Theta, p:B, q:D}{p \to q : \,\&\,(G_1, G_2) \vDash \Theta, p:A \oplus^{q} B, q:C \,\&^{p}\, D} \oplus \& \qquad \frac{G \vDash p:A, q:B}{?p \to !q : \langle G \rangle \vDash p:?A, q:!B} \, !?$$

Building on coherence, we generalize the cut-rule Cut to a multi-cut rule MCut that defines the type of a multi-party computation by combining multiple processes P_i.

$$\frac{P_1 \vdash \Gamma_1, x^{p_1} : A_1 \quad \ldots \quad P_n \vdash \Gamma_n, x^{p_n} : A_n}{(\nu x : G)\,(P_1 \mid \ldots \mid P_n) \vdash \Gamma_1, \ldots, \Gamma_n} \ \mathsf{MCut}, \text{ where } G \vDash p_1 : A_1^{\perp}, \ldots, p_n : A_n^{\perp}$$

In conclusion, this work presents to our knowledge the first formulation of a logic of multi-party sessions. The logic is a conservative extension over Wadler's CP, it is expressive as this example shows, and it is sound as cut-elimination and multi-cut elimination hold.

References

[CDCYP15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 760:1–65, 2015.

[CMSY15] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 412–426, 2015.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, volume 43(1), pages 273–284. ACM, 2008.

[Wad14] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.

Normalization by Evaluation in the Delay Monad

Andreas Abel[1] and James Chapman[2]

[1] Chalmers and Gothenburg University, Sweden
andreas.abel@gu.su
[2] University of Strathclyde, Glasgow, Scotland
james.chapman@strath.ac.uk

We present an Agda formalization of a normalization proof for simply-typed lambda terms. The normalizer consists of two coinductively defined functions in the delay monad: One is a standard evaluator of lambda terms to closures, the other a type-directed reifier from values to η-long β-normal forms. Their composition, normalization-by-evaluation, is shown to be a total function a posteriori, using a standard logical-relations argument. The normalizer is then shown to be sound and complete. The completeness proof proof is dependent on termination. We also discuss a variation on this normalizer where environments used by the evaluator contain delayed values which can be proven complete independently of termination using weak bisimilarity. This approach would be a realisation of an aim of this work to present a modular proof of normalization where termination, soundness and completeness are independent.

The successful formalization serves as a proof-of-concept for coinductive programming and reasoning using sized types and copatterns [3], a new and presently experimental feature of Agda [4].

Termination of a normalizer was described in [2]. The soundness and completeness proofs are new[1] and the alternative normalizer with delayed environments and accompanying normalization proof is ongoing work.

Delay Monad and potential non-termination. The delay monad [5] captures the idea of a computation that may return a value eventually or not at all. We represent functions that have not yet been proven terminating and are therefore untrusted as functions from values of type A to delayed computations of type $\mathsf{Delay}\ B$. Proving termination (asserting a basic level of trustworthiness) amounts to proving that for any input value the delayed computation will converge to a value. Given a constructive proof of termination one can derive a function from values of type A to values of type B.

Normalization algorithm. The normalization algorithm consists of two main components: (1) an evaluator that takes typed terms to intermediate values given an environment explaining the variables; and (2) a typed directed reifyer that takes intermediate values to syntact η-long β-normal forms. Neither component is apriori terminating but we can nonetheless combine them using monadic bind.

$$
\begin{array}{lcl}
\mathsf{eval} & : & \mathsf{Tm}\ \Gamma\ \sigma\ \to\ \mathsf{Env}\ \Delta\ \Gamma\ \to\ \mathsf{Delay}\ (\mathsf{Val}\ \Delta\ \sigma) \\
\mathsf{reify} & : & \mathsf{Val}\ \Delta\ \sigma\ \to\ \mathsf{Delay}\ (\mathsf{Nf}\ \Delta\ \sigma) \\
\mathsf{nf} & : & \mathsf{Tm}\ \Delta\ \sigma\ \to\ \mathsf{Delay}\ (\mathsf{Nf}\ \Delta\ \sigma) \\
\mathsf{nf}\ t & = & \mathsf{eval}\ \mathsf{id}\ t\ >\!\!>\!\!=\ \mathsf{reify}
\end{array}
$$

Normalization theorem. We prove three theorems about the normalization algorithm:

$$
\begin{array}{lcl}
\mathsf{termination} & : & \forall\,(t\ \ : \mathsf{Tm}\ \Delta\ \sigma) \to \exists\,(n : \mathsf{Val}\ \Delta\ \sigma).\ \mathsf{nf}\,t \Downarrow n \\
\mathsf{soundness} & : & \forall\,(t\ \ : \mathsf{Tm}\ \Delta\ \sigma) \to t\ \cong_{\beta\eta}\ \mathsf{nf}\ t \\
\mathsf{completeness} & : & \forall\,(t\ t' : \mathsf{Tm}\ \Delta\ \sigma) \to t\ \cong_{\beta\eta}\ t'\ \to\ \mathsf{nf}\ t\ \equiv\ \mathsf{nf}\ t'
\end{array}
$$

Decoupling soundess and completeness from termination amounts to a lifing of the soundness predicate and completeness relation to the the Delay monad, i.e., saying that the predicate/relation would hold eventually. In the relation case this is bisimilarity. For the algorithm specified above this is possible for soundness but not completeness. For a modified algorithm where environments contain delayed values completeness should also be possible but this presents technical challenges such as potentially moving to a sized version of the Delay monad which is not well supported by current versions of Agda and moving from reasoning up to equality to reasoning up to weak bisimularity.

References

[1] Andreas Abel and James Chapman. Normalization by evaluation in the delay monad: Formalization. http://github.com/andreasabel/continuous-normalization.

[2] Andreas Abel and James Chapman. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In Paul Levy and Neel Krishnaswami, editors, Proceedings 5th Workshop on *Mathematically Structured Functional Programming,* Grenoble, France, 12 April 2014, volume 153 of *Electronic Proceedings in Theoretical Computer Science*, pages 51–67. Open Publishing Association, 2014.

[3] Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:61, 2016. ICFP 2013 special issue.

[4] AgdaTeam. The Agda Wiki, 2016.

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

Towards Readable Program Correctness Proofs in Coq*

Jacek Chrząszcz and Aleksy Schubert[1]

University of Warsaw, [alx,chrzaszc]@mimuw.edu.pl

Abstract

Coq is a proof assistant that can be used in the process of program verification. We present a number of difficulties one can meet when Coq is used for this purpose and some techniques that can be employed to make the process of proving more comprehensive and accessible to formal proof developers.

Formal verification tools rely on providing detailed account of programs that are verified. All the details there must be mastered explicitly — the tools watch that no detail is overlooked in the implementation. The advantage of this approach is that they can be exploited in scenarios in which programmers should demonstrate their full understanding of code. The disadvantage is that the semantics of the program must be expressed in the native formalism of the tool and the translation itself is complicated. It is so complicated that the usual result is that the final description hardly resembles the original code and a considerable expertise is necessary to understand which particular nuance of the original code is currently being proved.

Even though the progress in automated theorem proving is significant there are and *will be* cases when one has to rely on interactive provers such as Coq to obtain verified piece of code. Actually, the interactive provers are inevitable when there is a mismatch between specification and code. They are difficult to replace when systematic search of mismatch is attempted. We propose to actively address the issues associated with program verification through development of techniques that help proof developers to construct their proofs in interactive environments. To make the process more concrete we demonstrate the usefulness of these techniques in the context of two program verification platforms, one of them being Frama-C coupled with Why3 and the other one HAHA (available from `http://haha.mimuw.edu.pl/`).

Frama-C is a set of tools that can be used to analyse correctness of programs written in the C programming language [1]. It generates Hoare-logic style verification conditions that can be given as input to Why3 and then one can manage their verification in Why3. In particular some of the verification conditions can be proven correct in Coq [2]. One can argue that verification conditions that are obtained from this tool-chain are real-world verification conditions that one should deal with in the process of real program verification.

HAHA is a tool that can be used to teach Hoare-logic basics through verification of programs written in a small imperative programming language. It is developed in Warsaw and there are initial results that indicate that its use can give advantage to students who have contact with it [3]. One of its features is the ability to generate verification conditions provable in Coq. The verification conditions obtained from this tool are simpler than those obtained from Frama-C. However, this has the advantage that they can serve better to illustrate solutions to problems encountered during verification.

In our talk we will compare the two environments and show most problematic situations that emerge during proof development. We divide the issues encountered into two basic categories. One is associated with structuring and presentation of the statements, the second is associated

*This work was partially supported by the Polish NCN grant no 2013/11/B/ST6/01381.

with proving mechanisms specific for program verification that can be made available in both of the environments to make proving efforts easier. In particular, the typical situation that needs to be handled is that due to some assignment a small portion of memory is changed, but some property must be carried over through the assignment. We developed a Coq tactic that is able to deal with such situation in many typical situations. We first did it for Coq scripts obtained from HAHA and then adapted to proofs done under Frama-C–Why tool-chain.

References

[1] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Proc. of SEFM'12*, volume 7504 of *LNCS*. Springer, 2012.

[2] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proc. of ESOP'13*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[3] Tadeusz Sznuk and Aleksy Schubert. Tool support for teaching hoare logic. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, volume 8702 of *LNCS*, pages 332–346, 2014.

Sprinkles of extensionality for your vanilla type theory

Jesper Cockx[1] and Andreas Abel[2]

[1] DistriNet – KU Leuven
[2] Department of Computer Science and Engineering – Gothenburg University

Dependent types can make your developments (be they programs or proofs) dramatically safer by allowing you to prove that you implemented what you intended. Unfortunately, they can also make your developments dramatically more boring by requiring you to elaborate those proofs in often painstaking detail. For this reason, dependently typed languages typically allow you to cheat by postulating some facts as axioms. However, type theory is not just about which types are inhabited; it's about how things *compute*. Computationally, an axiom is a stuck term, so it prevents you from evaluating your programs. What if you could postulate not just axioms, but also arbitrary rewrite rules?

1. A typical frustration for people new to proof assistants like Agda or Coq is that $0 + x$ evaluates to x for arbitrary x, but $x + 0$ doesn't. Of course, a lemma for $x + 0 = x$ is easy to prove, but having to appeal to this lemma explicitly in all subsequent uses is bothersome. By adding a rewrite rule $x + 0 \longrightarrow x$, you can get the automatic application of this lemma. Similarly, you can add a rule $x + \texttt{suc}\ y \longrightarrow \texttt{suc}\ (x + y)$, or $(x + y) + z \longrightarrow x + (y + z)$.

2. Allais, McBride, and Boutillier (2013) present a series of rewrite rules (which they call ν-rules) for functions on pairs and lists. For example, they have a rule for concatenating with an empty list ($l +\!\!+ \texttt{[]} \longrightarrow l$), but also rules for simplifying expressions involving `map` and `fold` (e.g. $\texttt{map}\ (\lambda x.\, x)\ l \longrightarrow l$).

3. Homotopy type theory (The Univalent Foundations Program, 2013) presents the concept of *higher inductive types*, inductive types that have not only regular constructors, but also path constructors that introduce additional equalities. However, current implementations of higher inductive types only have evaluation rules for applying a function to a regular constructor, but not for applying them to a path constructor. By adding rewrite rules to the eliminator of a higher inductive type, working with higher inductive types becomes easier and much more natural.

4. In observational type theory (Altenkirch, McBride, and Swierstra, 2007), the equality type $x =_A y$ is defined by case analysis on the type A. This can be emulated in other theories by a few custom rewrite rules, thus giving you the advantages of observational type theory, such as functional extensionality.

5. Custom rewrite rules also make it possible to define *shallow embeddings* of other languages in your language, making it possible to import developments into your own language without losing their computational properties. In fact, some languages like Dedukti (Boespflug, Carbonneaux, and Hermant, 2012) are built completely around this concept.

All these examples show how nice it can be to mix up the already sweet taste of vanilla intensional type theory with some extensional sprinkles in the form of rewrite rules. For this purpose we added a new facility to Agda, available from version 2.4.2.4 onwards with some improvements in 2.5.1, allowing you to specify proof-irrelevant rewrite rules that are plugged

into the evaluation mechanism of the typechecker. Concretely, you can declare the identity type to be the rewrite relation by the pragma {-# BUILTIN REWRITE _≡_ #-}, and then declare the proof plus0 : $x + 0 \equiv x$ to be a rewrite rule by the pragma {-# REWRITE plus0 #-}. By adding this lemma as a rewrite rule, it holds for arbitrary open terms, thus simplifying the definition of functions involving natural numbers in their types. For example, if you also add plusSuc : $(x\ y : \mathbb{N}) \to x + (\mathtt{suc}\ y) \equiv \mathtt{suc}\ (x + y)$ as a rewrite rule, then the following proof just works, instead of complaining that $x + 0 \neq x$ or $x + (\mathtt{suc}\ y) \neq \mathtt{suc}\ (x + y)$:

```
plus-comm : (x y : ℕ) → x + y ≡ y + x
plus-comm zero    y = refl
plus-comm (suc x) y = cong suc (plus-comm x y)
```

Our implementation of rewrite rules can also handle examples 2 and 3 without problems. We haven't tried 4 or 5 yet, but we see no reason why these examples wouldn't work as well.

By adding more definitional equalities, the size of your proof terms can be reduced quite drastically. Rewrite rules also allow advanced users to experiment with new evaluation rules, without actually modifying the typechecker. On the other hand, you shouldn't just use any kind of sprinkles. Even more than with axioms, rewrite rules can break the type system completely: not only soundness, but also confluence, termination, and even subject reduction are in danger. This means typechecking may become undecidable, and the typechecker may loop indefinitely. For example, adding a rewrite rule $x + y \longrightarrow y + x$ causes the typechecker to loop. It is possible (though quite difficult) to regain these properties by checking confluence, termination, and completeness of the added rewrite rules (Blanqui, 2005). An example of a fully developed system incorporating these checks is CoqMT (Strub, 2010). But no matter how good these checks are, there will always be cases where they are too restrictive and stand in the way of experimentation. So we'd rather let you decide which rewrite rules you want to add, as long as you promise to be extra careful. After all, sometimes it is more important to be able to experiment freely than it is to be 100% safe, and this is precisely the kind of situations we aim for. What can *you* do with the power of arbitrary rewrite rules? We invite you to try it for yourself!

References

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

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Programming languages meets program verification*, 2007.

Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 2005.

Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In *Second International Workshop on Proof Exchange for Theorem Proving*, 2012.

Pierre-Yves Strub. Coq modulo theory. In *Computer Science Logic*, 2010.

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

A formal language for cyclic operads

Pierre-Louis Curien[1] and Jovana Obradović[2]

[1] πr^2 team, IRIF, CNRS, Université Paris Diderot, and Inria, France
curien@pps.univ-paris-diderot.fr
[2] πr^2 team, IRIF, CNRS, Université Paris Diderot, Inria, France, and University of Novi Sad, Serbia
jovana@pps.univ-paris-diderot.fr

An operad is a collection of abstract operations of different arities, equipped with a notion of how to compose them and an action of permuting their inputs. Operads encode categories of algebras whose operations have multiple inputs and one output, such as associative algebras, commutative algebras, Lie algebras, etc. The interest in encoding more general algebraic structures was a part of the *renaissance of operads* in the early nineties of the last century, when various generalizations of operads came into existence. The formalism of cyclic operads was introduced by Getzler and Kapranov in [2]. The motivation came from the framework of cyclic homology: in their paper, Getzler and Kapranov show that, in order to define a cyclic homology for $\mathcal{O}$-algebras, $\mathcal{O}$ has to be what they call a cyclic operad. The enrichment of the (ordinary) operad structure is provided by adding to the action of permuting the inputs of an operation an action of interchanging its output with one of the inputs, in a way that is compatible with operadic composition.

The notion of a cyclic operad was originally given in the *unbiased* manner in [2, Definition 2.1], over the structure of a monad in a category of unrooted trees. These trees act as pasting schemes, and the operations decorating their nodes are "composed in one shot" through the structure morphism of the algebra. Like operads, *biased* cyclic operads can be defined by means of simultaneous compositions [2, Theorem 2.2] or of partial composition [5, Proposition 42]. The fact that two operations can now be composed by grafting them along wires that "used to be outputs" leads to another point of view on cyclic operads, in which they are seen as generalisations of operads for which an operation, instead of having inputs and an (exchangeable) output, now has "entries", and it can be composed with another operation along any of them. One can find such an *entries-only* definition in [4, Definition 48]. By contrast, we refer to the definitions based on describing cyclic operads as operads with extra structure as *exchangeable-output* ones.

The equivalence between the unbiased and biased definitions of a cyclic operad is formally given as a categorical equivalence that is, up to some extent, taken for granted in the literature. The issue that the construction of the structure morphism of an algebra over the monad out of the data of a biased cyclic operad should be shown independent of the way trees are decomposed has not been addressed in the proof of [2, Theorem 2.2], while the proof of [5, Proposition 42] is not given. Also, the monad structure is usually not spelled out in detail, in particular for what regards the correct treatment of the identities. The primary goal of this work is to formalise rigorously the equivalence between the unbiased and biased definitions of cyclic operads. Instead of comparing one of the two exchangeable-output biased definitions with the unbiased one, as done in [2, Theorem 2.2] and [5, Proposition 42], we show that the entries-only and the unbiased definition describe the same structure. Another particularity in our approach is that the appropriate categorical equivalence will be proved in a syntactical environment: a cyclic operad with biased composition will be expressed as a model of the equational theory determined by the axioms of the entries-only definition, while the monad of unrooted trees figuring in the unbiased approach will be expressed through a formal language called *μ-syntax*. Although μ-syntax was originally designed precisely to help us carry out this proof, it certainly has a value at the very level of encoding the (somewhat cumbersome) laws

of the partial composition operation for cyclic operads. In other words, we also propose it as an alternative representation of the biased structure of a cyclic operad. We see this work as an experiment in bringing syntactical and type-theoretical know-how in the formal study of other algebraic structures used in connection with higher categories.

The name and the language of the μ-syntax formalism are motivated by another formal syntactical tool, the $\mu\tilde{\mu}$-subsystem of the $\overline{\lambda}\mu\tilde{\mu}$-calculus, presented by Curien and Herbelin in [1]. In their paper, programs are described by means of expressions called commands, of the form

$$\langle \mu\beta.c_1 \,|\, \tilde{\mu}x.c_2 \rangle,$$

which exhibit a computation as the result of an interaction between a term $\mu\beta.c_1$ and an evaluation context $\tilde{\mu}x.c_2$, together with a symmetric reduction system

$$c_2[\mu\beta.c_1/x] \longleftarrow \langle \mu\beta.c_1 \,|\, \tilde{\mu}x.c_2 \rangle \longrightarrow c_1[\tilde{\mu}x.c_2/\beta],$$

reflecting the duality between call-by-name and call-by-value evaluation. In our syntactical approach, we follow this idea and view operadic composition as such a program, i.e. as an interaction between two operations f and g, where f provides an input β (selected with μ) for the output x of g (marked with $\tilde{\mu}$). By moving this concept to the entries-only setting of cyclic operads, the input/output distinction of the $\mu\tilde{\mu}$-subsystem goes away, leading to the existence of a *single binding operator* μ, whose purpose is to select the entries of two operations which are to be connected in this interaction. More precisely, the expressions of the μ-syntax are

$$c ::= \langle s \,|\, t \rangle \;\mid\; \underline{f}\{t_{x_i} \,|\, i \in \{1,\dots,n\}\} \qquad s,t ::= x \;\mid\; \mu x.c$$

typed as follows

$$\frac{}{\{x\} \,|\, x} \qquad \frac{f \in \mathcal{C}(\{x_1,\dots,x_n\}) \quad Y_{x_i} \,|\, t_{x_i} \text{ for all } i \in \{1,\dots,n\}}{\underline{f}\{t_{x_i} \,|\, i \in \{1,\dots,n\}\} : \bigcup_{i=1}^{n} Y_{x_i}} \qquad \frac{X \,|\, s \quad Y \,|\, t}{\langle s \,|\, t \rangle : X \cup Y} \qquad \frac{c : X \quad x \in X}{X \backslash \{x\} \,|\, \mu x.c}$$

and the equations are

$$\langle s \,|\, t \rangle = \langle t \,|\, s \rangle \qquad \langle \mu x.c \,|\, s \rangle = c[s/x] \qquad \mu x.c = \mu y.c[y/x] \qquad \underline{f}\{t_x \,|\, x \in X\} = \underline{f^\sigma}\{t_{\sigma(y)} \,|\, y \in Y\}$$

The action of putting in line the characterization of the monad of unrooted trees, built upon the formalism of *unrooted trees with half-edges* commonly used in the operadic literature, together with the characterization by means of μ-syntax, makes the greatest part of the work. It involves setting up an intermediate formalism of unrooted trees, called the formalism of *Vernon trees*, that provides concise and lightweight pasting shemes for cyclic operads, and whose syntactical flavour reflects closely the shape of normal forms of the μ-syntax. The formal characterisation of a Vernon tree captures precisely the information relevant for describing the corresponding monad, which eases the verifications of the appropriate laws.

References

[1] P. -L. Curien, H. Herbelin, The duality of computation, ACM SIGPLAN Notices, Volume 35 Issue 9, 233-243, September 2000.

[2] E. Getzler, M. Kapranov, Cyclic operads and cyclic homology, Geom., Top., and Phys. for Raoul Bott, International Press, Cambridge, MA, 167-201, 1995.

[3] M. Markl, Models for operads, Comm. Algebra, 24(4):14711500, 1996.

[4] M. Markl, Modular envelopes, OSFT and nonsymmetric (non-Σ) modular operads, arXiv:1410.3414.

[5] M. Markl, Operads and PROPs, arXiv:math/0601129.

[6] J. P. May, The geometry of iterated loop spaces, volume 271 of Lectures Notes in Mathematics. Springer-Verlag, Berlin, 1972.

Components of a Hammer for Type Theory: Coq Goal Translation and Reconstruction

Łukasz Czajka and Cezary Kaliszyk
University of Innsbruck, Austria
{lukasz.czajka,cezary.kaliszyk}@uibk.ac.at

Abstract

Proof assistants based on the Calculus of Constructions lack powerful general purpose automation. In this talk we present an extension of the various proof advice components to type theory: (i) a translation of a significant part of the Coq logic into the format of automated proof systems; (ii) improvements of the encoding for the goals practically arising in Coq proofs; as well as (iii) a reconstruction mechanism based on a Ben-Yelles-type algorithm combined with limited rewriting, congruence closure and a first-order generalization of the left rules of Dyckhoff's system LJT.

Formalizing proofs in interactive theorem provers based on the Calculus of Inductive Constructions and its extensions is a daunting task. For systems based on simpler foundations robust general purpose procedures are able to discharge many easier goals completely automatically [2], providing strong proof advice, which reduces human labor [7]. Providing such proof advice requires an encoding of type theory judgements and proof goals into the logics and formats of automated theorem provers (ATPs), as well as a reconstruction mechanism able to build type theory derivations based on ATP-found proofs. For higher-order logic both of these proof advice components have been studied thoroughly. Powerful encodings that are able to automatically derive properties of types, such as monotonicity, as part of the translation have been studied theoretically and implemented in the recent versions of proof advice tools HOL(y)Hammer [9] and Sledgehammer [10]. The built-in HOL automation is able to reconstruct the majority of the automatically found proofs using either internal proof search [8] or source-level reconstruction. The internal proof search mechanisms provided in Coq, such as the `firstorder` tactic [3], have been insufficient for this purpose so far. Matita's ordered-paramodulation [1] is able to reconstruct many goals with up to two or three premises, and the congruence-closure based internal automation techniques in Lean [5] are also promising.

In this talk, we present our recently developed proof advice components for type theory and systems based on it. We first introduce an encoding of the Calculus of Inductive Constructions, including the additional logical constructions introduced by the Coq system, in untyped first-order logic. Subsequently, we improve the encoding for the kind of goals that practically arise in Coq standard library formal proofs. Finally, we present a reconstruction mechanism based on a Ben-Yelles-type procedure combined with a first-order generalization of the left rules of Dyckhoff's LJT, congruence closure and heuristic rewriting.

The first part of the talk, partly based on [4], introduces an encoding of (a close approximation of) the Calculus of Inductive Constructions in untyped first-order logic. The encoding should be a practical one, which implies that its general theoretical soundness is not the main focus, instead it is important that the encoding is precise enough to provide useful information about the necessary proof dependencies or instantiations. For the sake of efficiency, terms of type Prop are encoded directly as FOL formulas using a function $\mathscr{F}$. Terms that have type Type but not Prop are encoded using a function $\mathscr{G}$ as guards which essentially specify what it means for an object to have the given type. For instance, $\forall f : \tau.\varphi$ where $\tau = \Pi x : \alpha.\beta$ is translated to $\forall f.\mathscr{G}(\tau, f) \to \mathscr{F}(\varphi)$ where $\mathscr{G}(\tau, f) = \forall x.\mathscr{G}(\alpha, x) \to \mathscr{G}(\beta, fx)$. So $\mathscr{G}(\tau, f)$ says that an object f has type $\tau = \Pi x : \alpha.\beta$ if for any object x of type α, the application fx has type β.

The encoding has been evaluated on the theorems proved in the Coq standard library, experimentally confirming that it is efficient and precise enough to build an automated reasoning-based proof advice system for Coq. In particular, state-of-the-art classical ATPs are able to reprove above 30% of the human-written proofs, and the dependencies used in the automatically found proofs are indeed those needed by the user to prove the theorems in Coq. We will discuss ideas how this number can be improved by adapting the techniques available in other proof assistants to type theory in the talk. In order to ultimately confirm the adequacy of the encoding, it is necessary to reconstruct the proofs in Coq.

In the second part of the talk, we report on an early work-in-progress on proof reconstruction. We evaluate the Coq internal reconstruction mechanisms including `tauto` and `firstorder` [3] on the original proof dependencies and on the ATP found proofs, which are in certain cases more precise. In particular `firstorder` seems insufficient for finding proofs for problems created using the advice obtained from the ATP runs. This is partly caused by the fact that it does not fully axiomatize equality, but even on problems which require only purely logical first-order reasoning its running time is often unacceptable.

The formulas that we attempt to reprove usually belong to fragments of intuitionistic logic low in the Mints hierarchy [11]. Most of proved theorems follow by combining a few known lemmas. The formulas are small and in practice often belong to the negative fragment of intuitionistic logic. This raises a possibility of devising an automated proof procedure optimized for this fragment and for the utilization of the advice obtained from the ATP runs. We implemented a preliminary version of a Ben-Yelles-type procedure (essentially `eauto`-type proof search with lightweight looping check) augmented with a first-order generalization of the left rules of Dyckhoff's system LJT [6], the use of the `congruence` tactic, and heuristic rewriting using equational hypotheses. An experimental evaluation on the problems originating from ATP proofs of lemmas in the Coq standard library shows that in our setting this algorithm tends to perform significantly better than the available Coq's tactics. Our tactic manages to reconstruct above 90% of the reproved theorems. However, it needs to be remarked that if we utilize the advice obtained from ATP runs then about 50% of the the reproved theorems follow by a combination of hypothesis simplification, the tactics `intuition`, `auto`, `easy`, `congruence` and a few simple heuristics. The reconstruction success rate of the `firstorder` tactic combined with various heuristics is about 70% if generic axioms for equality are added to the context.

References

[1] A. Asperti and E. Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Mathematical Knowledge Management (MKM 2007)*, volume 4573 of *LNCS*, pages 146–160. Springer, 2007.

[2] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

[3] P. Corbineau. First-order reasoning in the calculus of inductive constructions. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *LNCS*, pages 162–177. Springer, 2003.

[4] Ł. Czajka and C. Kaliszyk. Encoding the calculus of constructions in FOL for a hammer for Coq. Submitted, `http://cl-informatik.uibk.ac.at/cek/submitted/lcck-coqencode.pdf`, 2016.

[5] L. M. de Moura. Dependent type practice (invited talk). In J. Avigad and A. Chlipala, editors, *Conference on Certified Programs and Proofs (CPP 2016)*, page 2. ACM, 2016.

[6] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807, 1992.

[7] T. Hales. Developments in formal proofs. *Séminaire Bourbaki*, 1086, 2013–2014. abs/1408.6474.

[8] J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. D. Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.

[9] C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.

[10] L. C. Paulson and J. Blanchette. Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *International Workshop on the Implementation of Logics (IWIL 2010)*, volume 2 of *EPiC*, pages 1–10. EasyChair, 2012.

[11] A. Schubert, P. Urzyczyn, and K. Zdanowski. On the Mints hierarchy in first-order intuitionistic logic. In A. M. Pitts, editor, *Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, volume 9034 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2015.

Expressing theories in the $\lambda\Pi$-calculus modulo theory and in the DEDUKTI system

Ali Assaf[1], Guillaume Burel[2], Raphal Cauderlier[3], David Delahaye[4], Gilles Dowek[5], Catherine Dubois[2], Frédéric Gilbert[6], Pierre Halmagrand[3], Olivier Hermant[7], and Ronan Saillard[7]

[1] Inria and École polytechnique, ali.assaf@inria.fr
[2] ENSIIE, {guillaume.burel,catherine.dubois}@ensiie.fr
[3] Cnam and Inria, {raphal.cauderlier,pierre.halmagrand}@inria.fr
[4] Cnam and Université de Montpellier, david.delahaye@umontpellier.fr
[5] Inria and École Normale de Supérieure de Cachan, gilles.dowek@ens-cachan.fr
[6] École des Ponts, Inria, and CEA, frederic.gilbert@inria.fr
[7] MINES Paristech, {olivier.hermant,ronan.saillard}@mines-paristech.fr

Defining a theory, such as arithmetic, geometry, or set theory, in predicate logic just requires to chose function and predicate symbols and axioms, that express the meaning of these symbols. Using, this way, a single logical framework, to define all these theories, has many advantages.

First, it requires less efforts, as the logical connectives, $\wedge$, $\vee$, $\forall$... and their associated deduction rules are defined once and for all, in the framework and need not be redefined for each theory. Similarly, the notions of proof, model... are defined once and for all. And general theorems, such as the soundness and the completeness theorems, can be proved once and for all.

Another advantage of using such a logical framework is that this induces a partial order between theories. For instance, Zermelo-Fraenkel set theory with the axiom of choice (ZFC) is an extension of Zermelo-Fraenkel set theory (ZF), as it contains the same axioms, plus the axiom of choice. It is thus obvious that any theorem of ZF is provable in ZFC, and for each theorem of ZFC, we can ask the question of its provability in ZF. Several theorems of ZFC, that are provable in ZF have been identified, and these theorems can be used in extensions of ZF that are inconsistent with the axiom of choice.

Finally, using such a common framework permits to combine, in a proof, lemmas proved in different theories: if $\mathcal{T}$ is a theory expressed in a language $\mathcal{L}$ and $\mathcal{T}'$ a theory expressed in a language $\mathcal{L}'$, if A is expressed in $\mathcal{L} \cap \mathcal{L}'$, $A \Rightarrow B$ is provable in $\mathcal{T}$, and A is provable in $\mathcal{T}'$, then B is provable in $\mathcal{T} \cup \mathcal{T}'$.

Despite these advantages, several logical systems have been defined, not as theories in predicate logic, but as independent systems: Simple type theory, also known as Higher-order logic, is defined as an independent system—although it is also possible to express it as a theory in predicate logic. Similarly, Intuitionistic type theory, the Calculus of constructions, the Calculus of inductive constructions... are defined as independent systems. As a consequence, it is difficult to reuse a formal proof developed in an automated or interactive theorem prover based on one of these formalisms in another, without redeveloping it. It is also difficult to combine lemmas proved in different systems: the realm of formal proofs is today a tower of Babel, just like the realm of theories was, before the design of predicate logic.

The reason why these formalisms have not been defined as theories in predicate logic is that predicate logic, as a logical framework, has several limitations, that make it difficult to express modern logical systems.

1. Predicate logic does not allow the use of bound variables, except those bound by the quantifiers $\forall$ and $\exists$. For instance, it is not possible to define, in predicate logic, a unary function symbol $\mapsto$, that would bind a variable in its argument.

2. Predicate logic ignores the propositions-as-types principle, according to which a proof π of a proposition A is a term of type A.

3. Predicate logic ignores the difference between reasoning and computation. For example, when Peano arithmetic is presented in predicate logic, there is no natural way to compute the term 2×2 into 4. To prove the theorem $2 \times 2 = 4$, several derivation steps need to be used while a simple computation would have sufficed.

4. Unlike the notions of proof and model, it is not possible to define, once and for all, the notion of cut in predicate logic and to apply it to all theories expressed in predicate logic: a specific notion of cut must be defined for each theory.

5. Predicate logic is classical and not constructive. Constructive theories must be defined in another logical framework: constructive predicate logic.

This has justified the development of other logical frameworks, that address some of these problems. Problem 1 has been solved in extensions of predicate logic such as λ-Prolog and Isabelle. Problems 1 and 2 have been solved in an extension of predicate logic, called the Logical Framework, also known as the $\lambda\Pi$-calculus, and the λ-calculus with dependent types. Problems 3 and 4 have been solved in an extension of predicate logic, called Deduction modulo theory. Combining the $\lambda\Pi$-calculus and Deduction modulo theory yields the $\lambda\Pi$-calculus modulo theory, a variant of Martin-Löf's logical framework, which solves problems 1, 2, 3, and 4.

Problem 5 can also be solved in this logical framework.

In previous work, we have shown that all functional Pure type systems, in particular the Calculus of Constructions, could be expressed in this framework.

Our goal in this talk is twofold: first, we want to go further and show that other systems can be expressed in the $\lambda\Pi$-calculus modulo theory, in particular classical systems, logical systems containing a programming language as a subsystem such as FoCaliZe, simple type theory, and extensions of the Calculus of constructions with universes and inductive types. Second, we want to demonstrate this expressivity, not just with adequacy theorems, but also by showing that large libraries of formal proofs coming from automated and interactive theorem provers can be translated to and checked in Dedukti, our implementation of the $\lambda\Pi$-calculus modulo theory. To do so, we shall translate to Dedukti proofs developed in various systems: the automated theorem proving systems Zenon, Zenon modulo, iProver, and iProverModulo, the system containing a programming language as a subsystem, FoCaLiZe, and the interactive theorem provers HOL and Matita.

We show this way that the logical framework approach scales up and that it is mature enough, so that we can start imagining a single library of formal proofs expressed in different theories, developed in different systems, but formulated in a single framework.

The talk presents several examples of theories that have been expressed in Dedukti, and several large libraries of proofs expressed in these theories and translated to Dedukti. It presents ongoing work to express more theories, and to reverse engineer these proofs. Before that, it presents the $\lambda\Pi$-calculus modulo theory and the Dedukti system.

Computing with intuitionistic sequent proof terms: progress report

José Espírito Santo and Maria João Frade and Luís Pinto

University of Minho, Portugal

It is well-known that the formulation as sequent calculi of logics and type theories is to be preferred, if one is interested in a formalism suitable for proof search [13, 10]. But in type theories one needs not only to search for proofs and proof terms, but also to compute with them. Now it is not too controversial to say that not everything is understood regarding computation with proof terms in the sequent calculus format, as progress in the matter is still seen recently, e.g. either in the computational interpretation of cut-elimination on focused proofs [15, 1], or in the understanding of variables in sequent proof terms [4]. Hence "structural" matters about sequent proof terms still hinder the formulation of type theories as sequent calculi.

In the past decade two of the authors proposed and studied the system $\lambda\mathbf{Jm}$ as a vehicle for studying reduction procedures in the sequent calculus [6, 7]. The approach was modular, with the system designed to be as simple as possible, so that only the intricacies of the reduction procedures remained: the logic was the simplest one (intuitionistic implications as sole connective); the cut=redex paradigm [8, 2] was not followed, so that variables in proof terms could be treated as ordinary term variables [4]; substitution was treated as a meta-operation, with the corresponding cut-rule treated as an admissible typing rule, from which the call-by-name character of cut-elimination followed [2].

But, in order not to fall in mere natural deduction with generalized elimination [16, 9], actual use of the formulas in the l.h.s. of sequents was permitted in the inference process of $\lambda\mathbf{Jm}$. This entails, at the level of proof terms, the existence of a primitive mechanism of vectorization of arguments, familiar from the $\overline{\lambda}$-calculus [8]; and, at the level of reduction, not only that cut-elimination corresponds to the "multiary"[14] version of the $\beta\pi$-reduction found in λ-calculus with generalized applications [9], but also the existence of a second reduction process. The latter may be seen as the η-reduction for the $\tilde{\mu}$-operator [2], but is named μ-reduction, as in [14], where it was introduced.

Finally, a third process of reduction, most typical of sequent calculus, has been permanently considered in the study of $\lambda\mathbf{Jm}$: permutative conversion [3, 14]. The three reduction procedures were studied in isolation for their properties and computational interpretation, but also in their possible combinations. At some point, the authors attempted (in [5]) not only some systematization of the multiplicity of subsystems and kinds of normal forms that such combinations give rise to, but also some harmony out of the syntactic noise and explosion. Recent progress allows us to say now that we could have done better, and that is what we intend to report in this talk.

Following [12, 3], let us call *normal* a cut-free term that is irreducible for permutative conversion (the justification of the terminology is that normal terms are in bijection with normal natural deductions). Normal terms have a characterization that is applicable to terms in general (not necessarily cut-free). We call *natural* the terms which enjoy such characterization, so that a sequent proof term is normal iff it is natural and cut-free. Natural terms are closed for cut-elimination and μ-reduction. We can now give a simple and transparent computational interpretation of this subsystem: non-values consist of a term "applied" to a kind of generalized argument, consisting of a list of lists of ordinary arguments, with cut-elimination allowing the call of a function with the first argument of the first list (β rule), or appending two such lists

of lists (π rule), and with μ being an operation of flattening. So, the generality of generalized applications is here reduced to a second vectorization mechanism. In addition, we characterize proof search for normal proofs, identifying the relaxation of the LJT focusing discipline [8, 11] that it follows.

We also embrace the view that permutative conversion is a process of *conversion to natural form*, named γ for short, and amend the definition of γ found in [5] after having realized that the substitution process involved is a refinement of ordinary term substitution. Conversion to natural form is then studied systematically together with cut-elimination and μ-reduction to know when a procedure commutes and/or preserves another. Based on this analysis, we can conclude that a proof in $\lambda\mathbf{Jm}$ determines not one, but eight possibly distinct cut-free proofs. We also see how to combine γ-reduction and μ-reduction in order to define *focalization* - a process of reduction to the LJT-form - and observe the commutation of cut-elimination with focalization. Finally, since conversion to natural form commutes with cut-elimination, we see that the two immediate senses for the concept of *normalization* in $\lambda\mathbf{Jm}$, either conversion of cut-free terms to normal form, or cut-elimination in the natural subsystem, are coherent and have a common generalization to the entire set of proof terms.

References

[1] T. Brock-Nannestad, N. Guenot, and D. Gustafsson. Computation in focused intuitionistic logic. In *Proceedings of PPDP'15*, pages 43–54. ACM, 2015.

[2] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of International Conference on Functional Programming 2000*. IEEE, 2000.

[3] R. Dyckhoff and L. Pinto. Permutability of proofs in intuitionistic sequent calculi. *Theoretical Computer Science*, 212:141–155, 1999.

[4] J. Espírito Santo. Curry-Howard for sequent calculus at last! In *Proceedings TLCA'15*, volume 38 of *LIPIcs*, pages 165–179. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[5] J. Espírito Santo, M. J. Frade, and L. Pinto. Structural proof theory as rewriting. In *Proceedings of RTA 2006*, volume 4098 of *LNCS*, pages 197–211. Springer-Verlag, 2006.

[6] J. Espírito Santo and L. Pinto. Confluence and strong normalisation of the generalised multiary λ-calculus. In *Proceedings of TYPES'04 2003*, volume 3085 of *LNCS*. Springer-Verlag, 2004.

[7] J. Espírito Santo and L. Pinto. A calculus of multiary sequent terms. *ACM Trans. Comput. Log.*, 12(3):22, 2011.

[8] H. Herbelin. A λ-calculus structure isomorphic to a Gentzen-style sequent calculus structure. In *Proceedings of CSL'94*, volume 933 of *LNCS*, pages 61–75. Springer-Verlag, 1995.

[9] F. Joachimski and R. Matthes. Standardization and confluence for a lambda calculus with generalized applications. In *Proc. of RTA'00*, volume 1833 of *LNCS*, pages 141–155. Springer, 2000.

[10] S. Lengrand, R. Dyckhoff, and J. McKinna. A focused sequent calculus framework for proof search in pure type systems. *Logical Methods in Computer Science*, 7(1), 2011.

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

[12] G. Mints. Normal forms for sequent derivations. In P. Odifreddi, editor, *Kreiseliana*, pages 469–492. A. K. Peters, Wellesley, Massachusetts, 1996.

[13] D. Pym. A note on the proof theory of the lambda-π-calculus. *Studia Logica*, 54(2):199–230, 1995.

[14] H. Schwichtenberg. Termination of permutative conversions in intuitionistic Gentzen calculi. *Theoretical Computer Science*, 212, 1999.

[15] R. J. Simmons. Structural focalization. *ACM Trans. Comput. Log.*, 15(3):21:1–21:33, 2014.

[16] J. von Plato. Natural deduction with general elimination rules. *Annals of Mathematical Logic*, 40(7):541–567, 2001.

If-then-else and other constructive and classical connectives

Herman Geuvers and Tonny Hurkens

Radboud University & Technical University Eindhoven

Abstract

We develop a general method for deriving natural deduction rules from the truth table for a connective. The method applies to both constructive and classical logic. This implies we can derive "constructively valid" rules for any (classical) connective. We show this constructive validity by giving a general Kripke semantics, that is shown to be sound and complete for the constructive rules. For the well-known connectives, like $\vee$, $\wedge$, $\to$, the constructive rules we derive are equivalent to the natural deduction rules we know from Gentzen and Prawitz. However, they have a different shape, because we want all our rules to have a standard "format", to make it easier to define the notions of cut and to study proof reductions. In style they are close to the "general elimination rules" by Von Plato [4]. The rules also shed some new light on the classical connectives: e.g. the classical rules we derive for $\to$ allow to prove Peirce's law. Our method also allows to derive rules for connectives that are usually not treated in natural deduction textbooks, like the "if-then-else", whose truth table is clear but whose constructive deduction rules are not. We prove that "if-then-else", in combination with $\bot$ and $\top$, is functionally complete (all other constructive connectives can be defined from it). We define the notion of cut, generally for any constructive connective and we describe the process of "cut-elimination". Following the Curry-Howard isomorphism, we can give terms to deductions and we study cut-elimination as term reduction. We prove that reduction is strongly normalizing for constructive if-then-else logic.

Overview of the talk

Definition Suppose we have an n-ary connective c with a truth table t_c (with 2^n rows). We write $\phi = c(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are proposition letters and we write $\Phi = c(A_1, \ldots, A_n)$, where $A_1, \ldots, A_n$ are arbitrary propositions. Each row of t_c gives rise to an elimination rule or an introduction rule for c in the following way.

$$\begin{array}{cccc|c} p_1 & \ldots & p_n & \phi \\ \hline a_1 & \ldots & a_n & 0 \end{array} \quad \mapsto \quad \frac{\vdash \Phi \quad \ldots \vdash A_j \text{ (if } a_j = 1) \ldots \quad \ldots A_i \vdash D \text{ (if } a_i = 0) \ldots}{\vdash D}\,\text{el}$$

$$\begin{array}{cccc|c} p_1 & \ldots & p_n & \phi \\ \hline b_1 & \ldots & b_n & 1 \end{array} \quad \mapsto \quad \frac{\ldots \vdash A_j \text{ (if } b_j = 1) \ldots \quad \ldots A_i \vdash \Phi \text{ (if } b_i = 0) \ldots}{\vdash \Phi}\,\text{in}^i$$

$$\begin{array}{cccc|c} p_1 & \ldots & p_n & \phi \\ \hline c_1 & \ldots & c_n & 1 \end{array} \quad \mapsto \quad \frac{\Phi \vdash D \quad \ldots \vdash A_j \text{ (if } c_j = 1) \ldots \quad \ldots A_i \vdash D \text{ (if } c_i = 0) \ldots}{\vdash D}\,\text{in}^c$$

If $a_j = 1$ in t_c, then A_j occurs as a Lemma in the rule; if $a_i = 0$ in t_c, then A_i occurs as a Casus. The rules are given in abbreviated form and it should be understood that all judgments can be used with an extended hypotheses set Γ.

Example From the truth table we derive the following intuitionistic rules for $\wedge$, 3 elimination rules and one introduction rule:

$$\frac{\vdash A \wedge B \quad A \vdash D \quad B \vdash D}{\vdash D}\,\wedge\text{-el}_a \qquad \frac{\vdash A \wedge B \quad A \vdash D \quad \vdash B}{\vdash D}\,\wedge\text{-el}_b$$

$$\frac{\vdash A \wedge B \quad \vdash A \quad B \vdash D}{\vdash D}\,\wedge\text{-el}_c \qquad \frac{\vdash A \quad \vdash B}{\vdash A \wedge B}\,\wedge\text{-in}$$

These rules are all intuitionistically correct, as one can observe by inspection. We will show that these are equivalent to the well-known intuitionistic rules. We will also show how these rules can be optimized and be reduced to 2 elimination rules and 1 introduction rule.

From the truth table we also derive the following rules for $\neg$, 1 elimination rule and 1 introduction rule, a classical and an intuitionistic one.

$$\frac{\vdash \neg A \quad \vdash A}{\vdash D}\ \neg\text{-el} \qquad \frac{A \vdash \neg A}{\vdash \neg A}\ \neg\text{-in}^i \qquad \frac{\neg A \vdash D \quad A \vdash D}{\vdash D}\ \neg\text{-in}^c$$

As an example of the classical derivation rules we can show that $\neg\neg A \vdash A$ is derivable.

Contribution of the paper and related work

Natural deduction has been studied extensively, since the original work by Gentzen, both for classical and intuitionistic logic. Overviews can be found in [3] and [1]. Also the generalization of natural deduction to include other connectives or allow different derivation rules has been studied by various researchers. Notably, there is the work of Schroeder-Heister [2] and Von Plato [4] is related to ours. Schroeder-Heister studies general formats of natural deduction where also rules may be discharged (as opposed to the normal situation where only formulas may be discharged). He also studies a general rule format for intuitionistic logic and shows that the connectives $\wedge, \vee, \rightarrow, \bot$ are complete for it. Von Plato discusses "generalized elimination rules", which also appear naturally as a consequence of our approach of deriving the rules from the truth table.

However, we focus not so much on the rules but on the fact that we can define different and new connectives constructively. In our work, we derive the rules directly from the truth table and we give a complete Kripke semantics for the constructive connectives. This also allows us to prove some meta properties about the rules. For example, we give a generalization of the *disjunction property* in intuitionistic logic. We define and study cuts precisely, for the intuitionistic case. We look more in detail into the logic with just if-then-else and we prove that cut-elimination is strongly normalizing by studying the reduction of proof terms. We also show that if-then-else with $\bot$ and $\top$ is functionally complete for intuitionistic logic.

References

[1] S. Negri and J. von Plato. *Structural Proof Theory.* Cambridge University Press, 2001.

[2] P. Schroeder-Heister. A natural extension of natural deduction. *J. Symb. Log.*, 49(4):1284–1300, 1984.

[3] Dirk van Dalen. *Logic and structure (3. ed.).* Universitext. Springer, 1994.

[4] Jan von Plato. Natural deduction with general elimination rules. *Arch. Math. Log.*, 40(7):541–567, 2001.

A Type Theory for Comprehensive Parametric Polymorphism

Neil Ghani[1], Fredrik Nordvall Forsberg[1], and Alex Simpson[2]

[1] Department of Computer and Information Sciences, University of Strathclyde, UK
[2] Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Introduction

A polymorphic program is *parametric* if it applies the same uniform algorithm at all instantiations of its type parameters [14]. Reynolds [12] proposed *relational parametricity* as a mathematical model of parametric polymorphism. Relational parametricity is a powerful mathematical tool with many useful consequences [7, 15]. The polymorphic lambda-calculus $\lambda\mathbf{2}$ [6, 11] serves as a model type theory for (impredicative) polymorphism. Taken separately, the categorical structures needed to model $\lambda\mathbf{2}$ and relational parametricity are well-known ($\lambda\mathbf{2}$ *fibrations* [8, 13] and *parametricity graphs* [3, 4], respectively). However, simply combining these two notions results in a structure that enjoys the expected properties of parametricity only in the special case that the underlying category is *well-pointed*. Since well-pointedness rules out many categories of interest in semantics (e.g., functor categories) this limits the generality of the theory.

Existing solutions (e.g. Birkedal and Møgelberg [2]) overcome this restriction by adding significant additional structure to models (enough to model the full logic of Plotkin and Abadi [10]). We give instead a minimal solution, where we retain the original idea of combining reflexive graph categories with category-theoretic models of $\lambda\mathbf{2}$. We implement this in a perhaps unexpected way: *we modify the notion of* $\lambda\mathbf{2}$ *model* by asking for $\lambda\mathbf{2}$ fibrations to additionally satisfy Lawvere's *comprehension* property [9]. This way, we can interpret a combined context containing both type and term variables interspersed (similar to how contexts need to be handled in dependent type theories). We then combine such *comprehensive* $\lambda\mathbf{2}$ *fibrations* with parametricity-graph structure in order to also model relational parametricity. We call the resulting structures *comprehensive* $\lambda\mathbf{2}$ *parametricity graphs*.

A type theory for reasoning about parametricity

The main focus of this presentation is a type theory, $\lambda\mathbf{2R}$, which has a sound and complete interpretation in comprehensive $\lambda\mathbf{2}$ parametricity graphs, and which can be used to show that our models enjoy that the expected consequences of parametricity. This type theory is similar in many respects to System R of Abadi, Cardelli and Curien [1] and System P of Dunphy [3]. In addition to the standard judgements of System F, we add three new judgements: $\boxed{\Theta\ \mathsf{rctxt}}$ says that Θ is a well-defined *relational context*; $\boxed{\Theta \vdash A_1 R A_2\ \mathsf{rel}}$ says that R is a *relation* between types A_1 and A_2, in relational context Θ; and $\boxed{\Theta \vdash (t_1 : A_1) R (t_2 : A_2)}$ is a *relatedness judgement*, asserting that $t_1 : A_1$ is related to $t_2 : A_2$ by the relation R.

Importantly, both type and term variables on the left-hand side of relations are always manipulated separately from variables on the right. As a consequence, it does not make sense to talk about equality relations in the type theory, since there is no way to ensure that a judgement $\Theta \vdash (x : A) R (x : A)$ refers to "the same" x on the left and right hand side. For each $\Gamma \vdash A\ \mathsf{type}$, we instead define a *pseudo-identity relation* $\langle\Gamma\rangle \vdash A\langle A\rangle A\ \mathsf{rel}$ — the canonical relational interpretation of A. However, because of the changed context $\langle\Gamma\rangle$ (obtained by taking

the relational interpretation of Γ pointwise), $\langle A \rangle$ is not a proper identity relation in general — in fact, for open types B, the relation $\langle B \rangle$ is not even a homogeneous. The more subtle pseudo-identity property of $\langle A \rangle$ is instead given by the *parametricity rule*, which states that if $\langle \Gamma \rangle \vdash (s : A)\langle A \rangle(t : A)$, then $\Gamma \vdash s = t : A$. This rule is sound because of the parametricity-graph structure present in our models.

Graphs of functions are ubiquitous in standard arguments involving relational parametricity. Since we in general lack identity relations, we also lack graph relations, but we identify two forms of *pseudograph* relations, whose subtle interrelationship allows us to establish the consequences we need. One kind of pseudograph relation is immediately definable using the *fibrational* structure built into the notion of parametricity graph. The other type requires *opfibrational* structure. We use an impredicative encoding to show that opfibrational structure is definable in $\lambda\mathbf{2R}$. Finally, we can replay the usual proofs — but with graph relations replaced by pseudographs — showing that comprehensive $\lambda\mathbf{2}$ parametricity graphs enjoy the familiar consequences of parametricity.

Based on published work This is based on our recent paper published in FoSSaCS 2016 [5].

References

[1] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1&2):9–58, 1993.

[2] Lars Birkedal and Rasmus E. Møgelberg. Categorical models for Abadi and Plotkin's logic for parametricity. *Mathematical Structures in Computer Science*, 15:709–772, 2005.

[3] Brian Dunphy. *Parametricity as a notion of uniformity in reflexive graphs*. PhD thesis, University of Illinois, 2002.

[4] Brian Dunphy and Uday Reddy. Parametric limits. In *LICS*, pages 242–251, 2004.

[5] Neil Ghani, Fredrik Nordvall Forsberg, and Alex Simpson. Comprehensive parametric polymorphism: categorical models and type theory. In B. Jacobs and C. Löding, editors, *FoSSaCS 2016*, volume 9634 of *LNCS*. Springer, 2016.

[6] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris VII, 1972.

[7] Claudio Hermida, Uday Reddy, and Edmund Robinson. Logical relations and parametricity — a Reynolds programme for category theory and programming languages. *ENTCS*, 303:149–180, 2014.

[8] Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999.

[9] F William Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. *Applications of Categorical Algebra*, 17:1–14, 1970.

[10] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *LNCS*, pages 361–375. Springer, 1993.

[11] John Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *LNCS*, pages 408–425. Springer, 1974.

[12] John Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing*, pages 513–523, 1983.

[13] Robert A.G Seely. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic*, pages 969–989, 1987.

[14] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbolic Computation*, 13(1-2):11–49, 2000.

[15] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *FPCA*, pages 347–359. ACM, 1989.

Towards probabilistic reasoning about lambda terms with intersection types

Silvia Ghilezan[1], Jelena Ivetić[1], Zoran Ognjanović[2], and Nenad Savić[1]

[1] University of Novi Sad, Novi Sad, Serbia
{gsilvia | jelenaivetic | nsavic}@uns.ac.rs
[2] Mathematical Institute SASA, Belgrade, Serbia
zorano@mi.sanu.ac.rs

In the last 30 years several formal tools have been developed for reasoning about uncertain knowledge. One of these approaches concerns formalization in terms of probabilistic logics. Although the idea of probabilistic logic can be traced back to Leibnitz, Lambert and Boole, the modern development was started by Nils Nilsson, who tried to provide a logical framework for uncertain reasoning [7]. After Nilsson, a number of researchers proposed formal systems for probabilistic reasoning, for example [4], [5], [8].

Intersection types ([3]) were introduced in the lambda calculus as an extension of the simple types in order to overcome the limitations of the simple (functional) types. Indeed, lambda calculus with intersection types has two unique properties which do not hold in other type systems. First, it completely characterizes the termination of reduction, a.k.a strong normalization, in lambda calculus (e.g. [6]). Second, its type assignment is sound and complete with respect to the filter model, which was proven in the seminal paper by Barendregt et al. [1]. The latter result will be useful in this work.

We introduce in this paper a formal model $\mathsf{P}\Lambda^{\cap}$ for reasoning about probabilities of lambda terms with intersection types which is a combination of lambda calculus and probabilistic logic. We propose its syntax, Kripke-style semantics and an infinitary axiomatization. We first endow the language of typed lambda calculus with a probabilistic operator $P_{\geq s}$ and, besides the formulas of the form $M : \sigma$ and its Boolean combinations, we obtain formulas of the form

$$P_{\geq s} M : \sigma$$

to express that the probability that the lambda term M is of type σ is equal to or greater than s. More generally, formulas are of the form $P_{\geq s}\alpha$, where α is typed lambda statement $M : \sigma$ or its Boolean combination, so the following is a formula of our formal model as well:

$$[P_{=\frac{1}{3}}(x : \sigma \to \tau) \wedge P_{=\frac{2}{3}}(y : \sigma)] \Rightarrow [P_{=0}(xy : \tau) \vee P_{=\frac{1}{3}}(xy : \tau)].$$

We then propose a semantics of $\mathsf{P}\Lambda^{\cap}$ based on a set of possible worlds, where each possible world is a lambda model. The set of possible worlds is equipped with a probability measure μ. The set $[\alpha]$ is the set of possible worlds that satisfy the formula α. Then the probability of α is obtained as $\mu([\alpha])$. Finally, we give an infinitary axiomatization of $\mathsf{P}\Lambda^{\cap}$, which is a combination of deduction rules for lambda calculus with intersection types, axioms for the classical propositional logic, as well as the axioms for probabilistic logics, and prove the deduction theorem.

The main results we want to prove are the soundness and strong completeness of $\mathsf{P}\Lambda^{\cap}$ with respect to the proposed model, where strong completeness means that every consistent set of formulas is satisfiable. The construction of the canonical model is crucial for the proof and relays on two key facts. The first one is that the lambda calculus with intersection types is

complete with respect to the filter lambda model, [1], and the second one is the property that every consistent set can be extended to the maximal consistent set.

In the last decade, several probabilistic extensions of the λ-calculus have been introduced and investigated. They are concerned with introducing non-determinism and probabilities into the syntax and operational semantics of the λ-calculus in order to formalize computation in the presence of uncertainty rather than with providing a framework that would enable probabilistic reasoning about typed terms and type assignments.

A slightly similar approach to ours, that provides a framework for probabilistic reasoning about typed terms, was treated by Cooper et al. in [2], where the authors proposed a probabilistic type theory in order to formalize computation with statements of the form "a given type is assigned to a given situation with probability p". However, the developed theory was used for analyzing semantic learning of natural languages in the domain of computational linguistics, and no soundness or completeness issues were discussed.

References

[1] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symb. Log.*, 48(4):931–940, 1983.

[2] Robin Cooper, Simon Dobnik, Shalom Lappin, and Staffan Larsson. A probabilistic rich type theory for semantic interpretation. *Proceedings of the EACL 2014 Workshop on Type Theory and Natural Language Semantics (TTNLS)*, pages 72–79, 2014.

[3] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19(1):139–156, 1978.

[4] Ronald Fagin, Joseph Y. Halpern, and Nimrod Megiddo. A logic for reasoning about probabilities. *Inf. Comput.*, 87(1/2):78–128, 1990.

[5] M. Fattorosi-Barnaba and G. Amati. Modal operators with probabilistic interpretations, i. *Studia Logica*, 46(4):383–393, 1995.

[6] Silvia Ghilezan. Strong normalization and typability with intersection types. *Notre Dame Journal of Formal Logic*, 37(1):44–52, 1996.

[7] Nils J. Nilsson. Probabilistic logic. *Artif. Intell.*, 28(1):71–87, 1986.

[8] Zoran Ognjanovic and Miodrag Raskovic. Some probability logics with new types of probability operators. *J. Log. Comput.*, 9(2):181–195, 1999.

An imperative calculus for unique access and immutability

(extended abstract)

Paola Giannini[1], Marco Servetto[2], and Elena Zucca[3]

[1] Università del Piemonte Orientale, Italy
giannini@di.unipmn.it
[2] Victoria University of Wellington, New Zealand
marco.servetto@ecs.vuw.ac.nz
[3] DIBRIS, Università di Genova, Italy
elena.zucca@unige.it

We present a typed imperative calculus where it is possible to express and check aliasing and immutability property directly on source terms, without introducing invariants on an auxiliary structure which mimics physical memory. Indeed, continuing previous work [1, 5], we adopt an innovative model for imperative languages which, differently from traditional models, is a *pure calculus*. That is, execution is modeled by just rewriting source code terms, in the same way lambda calculus models functional languages. Formally, this is achieved by a non standard semantics of local variable declarations. When the expression defining a local variable x is evaluated, x is not replaced by its value but, rather, the association from x to the value is kept[1], and plays the role of an association from a reference to a (right) value in the store.

For instance, in the following code, where we assume a class `B` with a field of type `B`:

```
mut B x = new B(y)  mut B y = new B(x)  y
```

the two declarations can be seen as a store where `x` denotes an object of class `B` whose field is `y`, and conversely. Moreover, as shown in the example, variables (references) can be tagged by *modifiers* which specify constraints on their behaviour. Indeed, in the recent years a massive amount of research, see, e.g., [3, 4, 2] has been devoted to make programming with side-effects easier to maintain and understand, notably using type modifiers to control state access. Here, we consider two properties of references: (1) *no mutation*, that is, the reachable object graph denoted by the reference cannot be modified, (2) *no aliasing*, that is, the reachable object graph denoted by the reference cannot be saved as part of another object. We will use four modifiers corresponding to the four combinations, that is: mutation and aliasing (`mut`), no mutation (`imm` for immutable), no aliasing (`lent`), no mutation and no aliasing (`read`). In addition, we also use a `capsule` type modifier, a subtype of both the mutable and immutable modifiers, which allows mutable data to be passed and stored as internal state for an object without allowing other access to the same data.

Store is not flat, as it usually happens in models of imperative languages. For instance in the following example, where we assume a class `D` with an integer field, and a class `A` with two fields of type `B` and `D`, respectively:

```
imm D z = new D(0)
imm A w = {
  mut B x = new B(y)
  mut B y = new B(x)
    new A(x, z)
  }
w
```

[1] As it happens, with different aims and technical problems, in cyclic lambda calculi.

the store associates to `w` a block introducing local declarations, that is, in turn a store. In this representation, the fact that an object is not referenced from outside some enclosing object is directly modeled by the block construct: for instance, the object denoted by `y` can only be reached through `w`. Conversely, references from an object to the outside are directly modeled by free variables: for instance, the object denoted by `w` refers to the external object `z`.[2] In other words, our calculus smoothly integrates memory representation with shadowing and α-conversion.

We outline now the type system. A type T consists in a class name C decorated by a modifier μ.

The subtyping relation is the reflexive and transitive relation on types induced by

$$\mu\, C \leq \mu'\, C \text{ if } \mu \leq \mu'$$
$$\texttt{capsule} \leq \texttt{mut} \leq \texttt{lent} \leq \texttt{read}$$
$$\texttt{capsule} \leq \texttt{imm} \leq \texttt{read}$$

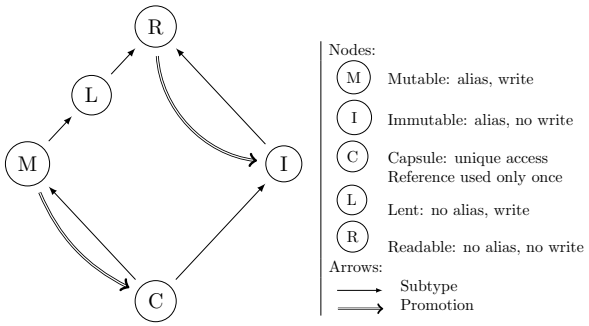


Figure 1: Type modifiers and their relationships

However, *promotion* rules can be used to move the type of an expression against the subtype hierarchy. More precisely: (1) Mutable expressions can be promoted to capsule, if mutable references are *weakly locked*, that is, can only be used as lent. (2) Readable expressions can be promoted to immutable, if lent, readable, and mutable references are *strongly locked*, that is, cannot be used at all. The situation is graphically depicted in Figure 1.

Our contribution includes the definition of the calculus with its type system, and proof of *soundness*. In addition, we formally stated and proved that modifiers imply their expected properties.

As a long term goal, we also plan to investigate (a form of) Hoare logic on top of our model. We believe that the hierarchical structure of our memory representation should help local reasoning.

References

[1] Andrea Capriccioli, Marco Servetto, and Elena Zucca. An imperative pure calculus. *ICTCS 2015*.

[2] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. *AGERE! 2015*, pages 1–12. ACM Press.

[3] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. *OOPSLA 2012*, pages 21–40. ACM Press.

[4] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. *POPL 2012*, pages 557–570. ACM Press.

[5] Marco Servetto and Elena Zucca. Aliasing control in an imperative pure calculus. *APLAS 2015*, *LNCS* 9458, pages 208–228. Springer.

[2] Note that the object denoted by `w` is a *capsule*, since its only external reference is `imm`, whereas its mutable state is encapsulated.

No value restriction is needed for algebraic effects and handlers *

Ohad Kammar[1], Sean K. Moss[2], and Matija Pretnar[3]

[1] University of Cambridge Computer Laboratory
ohad.kammar@cl.cam.ac.uk
[2] University of Cambridge Department of Pure Mathematics and Mathematical Statistics
s.k.moss@dpmms.cam.ac.uk
[3] University of Ljubljana, Faculty of Mathematics and Physics
matija.pretnar@fmf.uni-lj.si

The proposed talk describes submitted [7] and ongoing [6] work about the interaction of computational effects and predicative polymorphism. ML-style reference cells are known to be hard to combine with polymorphism [11, 4, 14], where a naïve type system is unsound (cf. Figure 1). The working solution, the value restriction [18] and its relaxation [3], are ad-hoc and restrict the programmer unnecessarily. We reexamine this problem in the context of algebraic effects [12] which extend the monadic account of computational effects [10] (e.g., the state monad) with the syntactic operations involving them (e.g., memory look-up and update). Bauer and Pretnar [2] use effect handlers, a generalisation of exception handlers that allows to handle arbitrary user-defined algebraic effects, to structure impure functional code, in analogy with monads [17]. The smooth integration of algebraic effects with polymorphism is surprising as effect handlers can implement local-state-like programming examples by manipulating continuations (k below) [13]:

$$\begin{array}{l} (\textbf{with } H_{ST} \textbf{ handle } \mathsf{set}\ \textbf{true}; \\ \qquad\qquad\qquad\quad \textbf{let } y = \mathsf{get}\ () \textbf{ in} \\ \qquad\qquad\qquad\quad \textbf{return } y)\ \textbf{false} \\ \rightsquigarrow^*\ \textbf{return true} \end{array} \qquad \text{where: } H_{ST} := \begin{array}{l} \textbf{handler } \{ \\ \textbf{return } x \mapsto \textbf{fun } \_ \mapsto \textbf{return } x \\ \mathsf{get}(\_; k) \mapsto \textbf{fun } s \mapsto k\ s\ s \\ \mathsf{set}(s'; k) \mapsto \textbf{fun } \_ \mapsto k\ ()\ s'\} \end{array}$$

$$\begin{array}{l} \textbf{let } r = \textbf{ ref } [\,] \textbf{ in} \\ r := [()]; \\ \textbf{true} :: !r \end{array}$$

Figure 1: unsound polymorphism and references [3]

In this work, we extend Bauer and Pretnar's [2] calculus for algebraic effects and handlers with Hindley-Milner polymorphism, in a standard way, without any value restriction:

- We add local effect signatures [8] Σ as finite mappings from operations op to pairs of value types A, B, which we denote by $(\mathsf{op} : A \to B) \in \Sigma$.
- We extend types with *type variables* α.
- We introduce *schemes* $\forall\vec{\alpha}.A$, where $\vec{\alpha}$ denotes a finite set of $|\vec{\alpha}|$-many type variables ranged over by α_i.

Our main result concerns the soundness of the type system w.r.t. the reduction relation $\rightsquigarrow$:

Theorem (Safety)**.** *If* $\vdash c : A\,!\,\Sigma$ *holds, then either: (i)* $c \rightsquigarrow c'$ *for some* $\vdash c' : A\,!\,\Sigma$*; (ii)* $c =$ **return** v *for some* $\vdash v : A$*; or (iii)* $c = \mathsf{op}(v; y.\, c')$ *for some* $(\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}) \in \Sigma$, $\vdash v : A_{\mathsf{op}}$*, and* $y : B_{\mathsf{op}} \vdash c' : A\,!\,\Sigma$*. In particular, when* $\Sigma = \emptyset$*, evaluation will not get stuck before returning a value.*

*Supported by the European Research Council grant 'events causality and symmetry — the next-generation semantics', and the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.

We use Leroy's [9] benchmarks for evaluating the interaction of effects and polymorphism. For example, if we extend the language with lists and bounded iteration, we can integrate effects in polymorphic functions, as for any Σ:

$$\begin{array}{l}\textbf{let}\ \mathrm{imp\_map} = \textbf{fun}\ f\ xs \mapsto \\ \quad \textbf{with}\ H_{ST}\ \textbf{handle}\ \Big(\Big(\textbf{foldl}\ \ \big(\textbf{fun}\ x \mapsto \mathsf{set}(f\ x :: \mathsf{get}\ ())\big)\ \ ()\ xs\Big)\ ;\ \mathrm{reverse}(\mathsf{get}\ ()) \\ \quad [\,]\ (*\ \text{initial state}\ *)\ \textbf{in}\ \ldots \qquad (*\ \mathrm{imp\_map} : \forall\alpha\beta.(\alpha \to \beta\,!\,\Sigma) \to (\alpha\ \mathsf{list} \to \beta\ \mathsf{list}\,!\,\Sigma)\,!\,\emptyset\ *)\end{array}$$

These benchmarks also highlight the limited expressiveness of effect handlers — we do not know how to implement, using effect handlers, even Leroy's basic benchmark, in which we return a newly allocated reference cell. The advantage is that, when trying to express the problematic program in Figure 1, we cannot express the first line, and the type system forbids handling the last two lines using the same state handler, as they refer to memory cells of different types.

A deeper soundness result comes from a denotational model for algebraic effects and polymorphism [6]. We modify Seely's models of impredicative polymorphism [15] by separating the fibred category of types into a fibred embedding of a fibred category of types into a fibred category of schemes. The universal quantifier $\forall$, previously right adjoint to structural weakening, is now replaced by a *relative* right adjoint [16, 1] along the inclusion of types in schemes. Using this relativisation, we can construct a parametric version of Harper and Mitchell's [5] set-theoretic models relative to a universal set $\mathcal{U}$. To add computational effects to this model, we construct a free fibred monad T_Δ and prove the following theorem, which allows us to interpret the calculus of algebraic effects and handlers, establishing soundness via a denotational model.

Theorem. *If $\mathcal{U} \neq \emptyset$, then the canonical morphism $T_{\Delta'}\forall\Delta.\tau \to \forall\Delta.T_{\Delta'\times\Delta}\tau$ is invertible.*

References

[1] Altenkirch, T., Chapman, J., and Uustalu, T. (2014). Monads need not be endofunctors. *CoRR*, **abs/1412.7148**.

[2] Bauer, A. and Pretnar, M. (2015). Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, **84**(1), 108–123.

[3] Garrigue, J. (2004). Relaxing the value restriction. In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer Berlin Heidelberg.

[4] Harper, R. and Lillibridge, M. (1993). Polymorphic type assignment and CPS conversion. *Lisp and Symbolic Computation*, **6**(3-4), 361–380.

[5] Harper, R. and Mitchell, J. C. (1993). On the type structure of standard ml. *ACM Trans. Program. Lang. Syst.*, **15**(2), 211–252.

[6] Kammar, O. and Moss, S. K. (2015). A denotational semantics for Hindley-Milner polymorphism. talk given at the 4th ACM SIGPLAN Workshop on Higher-Order Programming with Effects.

[7] Kammar, O. and Pretnar, M. (2015). No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming*. Submitted.

[8] Kammar, O., Lindley, S., and Oury, N. (2013). Handlers in action. *SIGPLAN Not.*, **48**(9), 145–158.

[9] Leroy, X. (1992). *Typage polymorphe d'un langage algorithmique*. Phd thesis (in french), Université Paris 7.

[10] Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, **93**(1), 55–92.

[11] Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press, Cambridge, MA, USA.

[12] Plotkin, G. D. and Power, J. (2002). Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer Berlin Heidelberg.

[13] Plotkin, G. D. and Pretnar, M. (2013). Handling algebraic effects. *Logical Methods in Computer Science*, **9**(4).

[14] Rémy, D. (2015). Type systems. Lecture notes, Parisian Master of Research in Computer Science.

[15] Seely, R. A. G. (1987). Categorical semantics for higher order polymorphic lambda calculus. *J. Symb. Log.*, **52**(4), 969–989.

[16] Ulmer, F. (1968). Properties of dense and relative adjoint functors. *Journal of Algebra*, **8**(1), 77 – 95.

[17] Wadler, P. (1990). Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, New York, NY, USA. ACM.

[18] Wright, A. K. (1995). Simple imperative polymorphism. *LISP and Symbolic Computation*, **8**(4), 343–355.

Non-Recursive Truncations

Nicolai Kraus

University of Nottingham
nicolai.kraus@nottingham.ac.uk

Abstract

I discuss non-recursive higher inductive types in homotopy type theory, and universal properties of truncations with respect to arbitrary types.

Homotopy type theory, often abbreviated as *HoTT*, is a branch of intensional Martin-Löf type theory based on the observation that types can be interpreted as some form of spaces. Besides univalent universes, the so-called *higher inductive types* (HITs) are a major new concept which is considered in HoTT. These are a powerful generalisation of inductive types. A HIT has not only constructors which define elements (*point constructors*), it may also have constructors that define equalities (*higher* or *path constructors*). A popular example is the circle $\mathbb{S}^1$, which we can represent by a point constructor $\mathsf{base} : \mathbb{S}^1$ and a path constructor $\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$. Another seemingly innocent HIT is the one implementing the *propositional truncation*: For any type A, we have the HIT $\|A\|$ with one point constructor $|-| : A \to \|A\|$ and one path constructor $\mathsf{t} : \Pi_{x,y:\|A\|}\, x = y$. The propositional truncation is probably the most prominent concept that can (non-trivially) be implemented as a HIT (see e.g. Awodey's and Bauer's *bracket types* in extensional type theory [1]). The type $\|A\|$ represents the *proposition* (i.e. has at most one inhabitant) that A is inhabited without requiring a concrete element of A to be specified. Many more examples of HITs can be found in the standard reference [5].

The HIT $\mathbb{S}^1$ is somewhat easy to visualise geometrically: first, we have a point; and second, we have a path from this point to itself. This is a particularly easy example of a finitely represented CW-complex (which already form a very well-behaved class of spaces themselves). The universal property of $\mathbb{S}^1$ says that, for any type X, functions $\mathbb{S}^1 \to X$ correspond exactly to pairs $\Sigma\,(x : X)\,.\,x = x$.

General HITs are much harder to understand because of the higher *inductive* component. We say that $\mathbb{S}^1$ is a *non-recursive* HIT because no constructor quantifies over elements of $\mathbb{S}^1$ itself. In the presentation of $\|A\|$ that we have given, the constructor t does quantify over elements of $\|A\|$ itself. This has the consequence that the induction and recursion principle, and the universal property, are somewhat restricted: a priori, we know that functions $\|A\| \to X$ correspond to functions $A \to X$, but only if X is propositional itself (i.e. fulfils the condition posed by the constructor t).

The described situation gives rise to two related questions for a given ("recursive") HIT H. First, can we formulate a version of H's universal property (or elimination principle) which is applicable when eliminating into *any* type? And, second, can H be represented as a non-recursive HIT, and does this give rise to a *useful* elimination principle into general types?

These questions are unsolved for a general H, but some answers have been found for the canonical example – the propositional truncation. In this talk, I want to compare three different constructions and results on this topic. The first can be found in my own article [3] (TYPES'14 post-proceedings), where I have shown a correspondence between *coherently constant functions* $A \to B$ and functions $\|A\| \to B$. The second is van Doorn's construction of the propositional truncation as a non-recursive HIT [6] (CPP'16). The third is my construction of the propositional truncation as a non-recursive HIT using a sequence of actual *approximations* [4] (to appear at LICS'16). In some more detail, the three mentioned constructions can be described as follows:

1. In [3], I have constructed *coherently constant functions* as morphisms between type-valued presheaves. In type-theoretic notation, a coherently constant function is given by an infinite tower of components, starting with: on level $[0]$, a function $f : A \to B$; on level $[1]$, a proof c of "weak constancy", i.e. $c : \mathsf{wconst}_f :\equiv \Pi_{a_1,a_2:A} f(a_1) = f(a_2)$; on level $[2]$, a proof of coherence for c, in the sense of $\mathsf{coh}_{f,c} :\equiv \Pi_{a_1,a_2,a_3:A}(a_1, a_2) \cdot c(a_2, a_3) = c(a_1, a_3)$; and so on.
   In the special case where B is n-truncated, all but the first $(n+2)$ components become trivial and can be omitted. This gives a reasonably clean characterisation of maps $\|A\| \to B$. We can reformulate this finite case as follows: consider the HIT $H^n(A)$ with a constructor $f : A \to H^n(A)$, a constructor $c : \mathsf{wconst}_f$, and so on ($n+2$ constructors). Then, we have the equivalence $\|H^n(A)\|_n \simeq \|A\|$. Unfortunately, we cannot write down the "full" HIT with infinitely many constructors.
2. Van Doorn has constructed $\|A\|$ as a non-recursive HIT [6]. The idea is, from my point of view, that we can simply take $H^0(A)$, i.e. drop all but the first two constructors, and make up for this by iterating H^0 infinitely often. This can be expressed as a colimit over a graph and is thus internal. A shortcoming is that this construction is not well-behaved if the iteration is done only finitely many times; as an example, already $H^0(H^0(1))$ is hard to visualise. In particular, $\left\|H^0(\ldots(H^0(A))\ldots)\right\|_n$ is nearly never equivalent to $\|A\|$.
3. My construction in [4] is an attempt to internalise an idea of [3]. At the same time, it looks similar to van Doorn's construction in that it is the colimit of a sequence. The difference is that van Doorn's construction forces everything to become equal in each step (which is much more than necessary), while mine only forces everything in some higher path space to become equal, depending on which step we are at. This results in a sequence of actual approximations of $\|A\|$, in the sense that the "connectedness level" increases in every step. The condition of the derived elimination principle is easier to satisfy than van Doorn's, from which we get a concrete consequence for the finite cases in [6]. It allows the formulation of an elimination principle for k-truncations into n-types, generalising the main result of [2].

References

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

[2] Paolo Capriotti, Nicolai Kraus, and Andrea Vezzosi. Functions out of higher truncations. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic (CSL) 2015)*, volume 41 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 359–373, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] Nicolai Kraus. The general universal property of the propositional truncation. In Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau, editors, *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 111–145, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] Nicolai Kraus. Constructions with non-recursive higher inductive types. To appear in the proceedings of LICS'16, 2016.

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

[6] Floris van Doorn. Constructing the propositional truncation using non-recursive hits. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 122–129, 2016.

On the Set Theory of Fitch-Prawitz*
ABSTRACT

Furio Honsell[1], Marina Lenisa[1], Luigi Liquori[2], Ivan Scagnetto[1]

[1] Università di Udine, Italy, {*furio.honsell, marina.lenisa, ivan.scagnetto*}*@uniud.it*
[2] INRIA, France, *Luigi.Liquori@inria.fr*

The discovery of Russell's and Curry's Paradoxes inhibited mainstream research from exploring self-referential and inclusive Foundational Theories. The very concept of *type* was introduced precisely to disallow self-application, thus preventing itself from providing a *theory of everything*. In the last century, however, there have been isolated attempts to put forward some inclusive set theories. In 1952 Frederic B. Fitch [1] introduced a consistent Set-Theory, which compensates the effects of un-constrained abstraction by restricting the class of proofs. He introduced two possible restrictions which are rather idosyncratic and too restrictive[1]. It was not until Prawitz [7] gave a *natural deduction* presentation of Fitch's Theory, that a principled restriction on proofs was introduced, namely that the proof *be normal*.

Apart from this restriction on proofs, Fitch-Prawitz Set Theory, FP, is a standard first order theory with classical negation. Sets, *i.e.* abstractions, are introduced and eliminated in the natural way, and equality is expressed by *Leibniz equality*. The crucial rules are:

$$\lambda\mathrm{I})\quad \frac{A[t/x]}{t \in \lambda x.A} \qquad \lambda\mathrm{E})\quad \frac{t \in \lambda x.A}{A[t/x]} \qquad \bot)\quad \frac{\begin{array}{c}(\neg A)\\ \vdots \\ \bot\end{array}}{A}$$

FP is provably consistent almost by definition. Since there is no introduction rule for $\bot$, it cannot be derived by a normal proof. Thus even if Russell's class, $R \equiv \lambda x.x \notin x$, can be defined, Russell's Paradox does not fire since the proof of contradiction is not *normalizable* and hence not valid. *Tertium non datur* holds but *Aristotle's non-contradiction pinciple* fails in that both $\vdash_{\mathsf{FP}} R \in R$ and $\vdash_{\mathsf{FP}} R \notin R$ are derivable. FP subsumes higher-order logic for all orders. A considerable part of the theory of real numbers can be developed in FP, although standard rules such as *modus ponens* or *extensionality* are not admissible.

The root reason for Russell's paradox is not *extensionality* or *tertium non datur*. Curry's paradox holds also in Minimal Logic with no use of extensionality. The two catastrophic ingredients are *unrestricted contraction* or *unrestricted proofs*. We get consistent theories when any of the two are guarded. Grishin [4] showed that extensionality implies contraction, but Girard in [3] showed that *Light Linear Logic*'s contraction yields a perfectly sensible, albeit weak Set-Theory. FP rules out contradictions by not allowing to introduce $\bot$ by brute force.

We discuss Fitch-Prawitz Set Theory *per se*, and give a *Fixed Point Theorem* whereby one can show that all provably total recursive functions are typable in FP.

Universal Set Theories which support extensionality have been occasionally introduced in the literature. One of the most inclusive is the Theory of *Hyperuniverses*, see [2]. Consistency is achieved by restricting the class of predicates which are allowed in the λ-*abstraction* rules, to *generalized positive formulæ*. These are defined as follows:

*A first version of this work was presented at C.O.M.FORT. 2015 - Workshop in honor of Marco Forti, Pisa, 22/05/2015.

[1] Fitch *simple restriction* does not validate $A \to (B \to ((A \wedge B) \to C) \to C))$, while *special restriction* does not permit $(P \leftrightarrow (P \to Q)) \to Q$ or $((A \to (A \to B) \to B) \to A) \to A$.

Definition 1 (Generalized Positive Formulæ)**.** *The* Generalized Positive Formulæ *are the smallest class of formulæ which*

- *include* $x \in y$, $x = y$*;*
- *are closed under the logical connectives* $\wedge, \vee$*;*
- *are closed under the quantifiers* $\forall x, \exists x, \forall x \in y, \exists x \in y$*;*
- *are closed under the formula* $\forall x.(\theta \to \phi)$*, where* ϕ *is a GPF and* $FV(\theta) = \{x\}$*.*

We provide an intriguing connection between FP and the Theory of Hyperuniverses. Namely we show that the strongly extensional collapse, *i.e. bisimilarity quotient*, of Fitch-Prawitz $\mathcal{P}$-coalgebra (V, f_{FP}), where V is the set of closed terms of FP and $f_{\mathsf{FP}}(t) = \{s \mid \vdash_{\mathsf{FP}} s \in t\}$, satisfies the astraction principle for *generalized positive formulæ*.

Finally, we show how to encode FP in the Logical Framework $\mathsf{LLF}_{\mathcal{P}}$ introduced in [6], using *monadic locks*. This work appears in [5]. $\mathsf{LLF}_{\mathcal{P}}$ is an extension of the Logical Framework LF which allows for delegating to an external tool the task of checking that a proof-term satisfies a given constraint. In the case of FP the constraint is that the proof term encodes a *normalizable proof*. This is, in fact, a slight generalization of the original system of Prawitz which allows for a *semi-decidable* notion of proof. The added value of using $\mathsf{LLF}_{\mathcal{P}}$ w.r.t. traditional LF is that the encoding can be shallower and hence more transparent. Taking `o` as the type of propositions and ι as the type of terms, the encoding requires to introduce two *judgements*, namely *valid* (`V:o->Type`) and *true* (`T:o->Type`). Only valid judgements can be assumed but only true judgements can be proved, whence a weaker form of $\to$-elimination can be expressed. For instance, if we consider the fragment of Fitch Prawitz Set Theory with $\to$ and the "membership" predicate ϵ as the constructors for propositions, we can introduce in the signature Σ_{FPST} the following constants:

`lam`: $(\iota$ `->` `o)->` ι $\quad$ ϵ: ι `->` ι `->` `o` $\quad$ $\to$: `o ->` `o ->` `o` $\quad$ δ: Π`A:o.(V(A) -> T(A))`

λ`_I`: Π`A:`ι `->o.`Π`x:`ι`.T(A x) -> T(`ϵ `x (lam A)) ` λ`_E`: Π`A:`ι `->o.`Π`x:`ι`.T(`ϵ `x (lam A)) -> T(A x)`

$\to$`_I : `Π`A,B:o.(V(A) -> T(B)) -> (T(A`$\to$`B))`

$\to$`_E : `Π`A,B:o.`Π`x:T(A).`Π`y:T(A`$\to$`B) -> `$\mathcal{L}^{\mathtt{Fitch}}_{\langle \mathtt{x},\mathtt{y}\rangle,\mathtt{T(A)}\times\mathtt{T(A}\to\mathtt{B)}}[\mathtt{T(B)}]$

where `lam` is the "abstraction" operator for building "sets", δ is the coercion function, and $\langle \mathtt{x}, \mathtt{y} \rangle$ denotes the encoding of pairs. The predicate in the lock $\mathtt{Fitch}(\Gamma \vdash_{\Sigma_{\mathsf{FPST}}} \langle \mathtt{x}, \mathtt{y} \rangle{:}\mathtt{T(A)}\times\mathtt{T(A} \to \mathtt{B)})$ holds iff `x` and `y` have *skeletons* in $\Lambda_{\Sigma_{\mathsf{FPST}}}$ (*i.e.*, in the set of $\mathsf{LLF}_{\mathcal{P}}$ terms definable using constants from the signature Σ_{FPST}), all the holes of which have either type `o` or are guarded by a δ, and hence have type `V(A)`, and, moreover, the proof derived by combining the *skeletons* of `x` and `y` is normalizable in the natural sense. Clearly, this predicate is only semidecidable.

References

[1] Frederic B. Fitch. *Symbolic logic*. New York, 1952.

[2] Marco Forti, Furio Honsell, and Marina Lenisa. Axiomatic Characterizations of Hyperuniverses and Applications. *Annals of the New York Academy of Sciences*, 806(1):140–163, 1996.

[3] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[4] Viacheslav Nikolaevich Grišin. Predicate and set-theoretic calculi based on logic without contractions. *Mathematics of the USSR-Izvestiya*, 18(1):41–59, 1982.

[5] Furio Honsell, Luigi Liquori, Petar Maksimović, and Ivan Scagnetto. Gluing together proof environments: Canonical extensions of lf type theories featuring locks. *arXiv preprint arXiv:1507.08051*, 2015.

[6] Furio Honsell, Luigi Liquori, Petar Maksimović, and Ivan Scagnetto. LLFP: A Logical Framework for modeling External Evidence, Side Conditions, and Proof Irrelevance using Monads. *Logical Methods in Computer Science*, 2016. To appear in a Special Issue in honor of P.-L. Curien.

[7] D. Prawitz. *Natural Deduction. A Proof Theoretical Study*. Almqvist Wiksell, Stockholm, 1965.

A Linear Dependent Type Theory*

Zhaohui Luo[1] and Yu Zhang[2]

[1] Royal Holloway, Univ of London
zhaohui.luo@hotmail.co.uk
[2] Institute of Software, Chinese Academy of Sciences
yzhang@ios.ac.cn

Introduction. Introducing linear types [2] (and, in general, resource sensitive types [6]) into a type theory with dependent types has been an interesting but difficult topic. One of the most difficult issues is whether to allow types to depend on linear variables. For instance, for $f : A \multimap A$, the equality type $Eq_A(f\ x,\ x)$ depends on the linear variable x of type A. In all of the existing research so far (see, for example, [1, 3, 5]), types are only allowed to depend on intuitionistic variables, but not on linear variables. In this paper, we present LDTT, a dependent type theory where types may depend on linear variables.

LDTT: a Linear Dependent Type Theory. A context in LDTT may contain two forms of entries: the usual (intuitionistic) entries $x{:}A$ and the linear ones $y{::}A$, where y is called a linear variable. For any term t, $FV(t)$ is the set of free variables occurring in t and, for any context Γ, if $FV(t) \subseteq FV(\Gamma)$, then $D_\Gamma(t)$ is the set of dependent variables of t w.r.t. Γ, defined as follows: (1) $FV(t) \subseteq D_\Gamma(t)$, and (2) for any $x \in D_\Gamma(t)$, $FV(\Gamma_x) \subseteq D_\Gamma(t)$. In LDTT, we have the following variable typing rule:

$$(V) \qquad \frac{\Gamma,\ x{:}\bar{:}A,\ \Gamma'\ \mathit{valid} \quad (\text{for all } y{::}\Gamma_y \in \Gamma,\ y \in D_\Gamma(x)) \quad \Gamma' \text{ intuitionistic}}{\Gamma,\ x{:}\bar{:}A,\ \Gamma' \vdash x : A} \quad (\bar{:} \in \{:, ::\})$$

where 'Γ' intuitionistic' means that Γ' does not contain any linear entries.

We have two forms of Π-types: the intuitionistic $\Pi x{:}A.B$ and linear $\Pi x{::}A.B$, whose rules are given in Figure 1. For both, the formation and introduction rules are not unusual, although their elimination rules need some explanations. For intuitionistic Π-types, in order to type $f(a)$ under context Γ, a is required to be typable in $\overline{\Gamma}$, the intuitionistic part of Γ, obtained from Γ by removing all the entries whose variables are in $FV_{LD}(\Gamma)$, the set of linear dependent variables in Γ.[1] The elimination rule for linear Π-types involves the operation $Merge(\Gamma;\Delta)$, which is only defined, notation $Merge(\Gamma;\Delta)\downarrow$, if $\overline{\Gamma} \equiv \overline{\Delta}$ and $FV_{LD}(\Gamma) \cap FV_{LD}(\Delta) = \emptyset$: (1) $Merge(\langle\rangle;\langle\rangle) = \langle\rangle$; (2) If $x \in FV_{LD}(\Gamma,\Delta)$, $Merge(\Gamma, x{:}\bar{:}A;\ \Delta) = Merge(\Gamma;\ \Delta, x{:}\bar{:}A) = Merge(\Gamma;\Delta), x{:}\bar{:}A$, where $\bar{:}$ is either : or ::; and (3) $Merge(\Gamma, x{:}A;\ \Delta, x{:}A) = Merge(\Gamma;\Delta), x{:}A$.

LDTT also contains equality types $Eq_A(a,b)$, whose rules are given in Figure 2. The Eq-formation rule involves another context merge operation $merge(\Gamma;\Delta)$, which is only defined, notation $merge(\Gamma;\Delta)\downarrow$, under the condition that, if $x{:}\bar{:}A \in \Gamma$ and $x{:}\bar{:}B \in \Delta$, then (1) $\bar{:}$ is either both : or both :: and (2) $A \equiv B$: (1) $merge(\Gamma;\langle\rangle) = \Gamma$, and (2) for $\bar{:}$ being either : or ::, $merge(\Gamma;\ x{:}\bar{:}A, \Delta)$ is equal to (i) $merge(\Gamma;\Delta)$, if $x \in FV(\Gamma)$, and (ii) $merge(\Gamma, x{:}\bar{:}A;\ \Delta)$, otherwise. Its elimination rule involves the $Merge$-operation defined earlier.

In linear logic, every linear variable occurs free for exactly once in a typed term. In LDTT, every linear variable *occurs essentially* for exactly once in a typed term – a property we call

*Partially supported by EU COST Action CA15123 and CAS/SAFEA International Partnership Program.

[1] Formally, $FV_{LD}(\Gamma)$ is defined as follows: (1) $FV_{LD}(\langle\rangle) = \emptyset$; (2) $FV_{LD}(\Gamma, x{::}A) = FV_{LD}(\Gamma) \cup \{x\}$; (3) if $FV(A) \cap FV_{LD}(\Gamma) = \emptyset$, then $FV_{LD}(\Gamma, x{:}A) = FV_{LD}(\Gamma)$; otherwise, $FV_{LD}(\Gamma, x{:}A) = FV_{LD}(\Gamma) \cup \{x\}$.

Intuitionistic Π-*types* (Conversion: $(\lambda x{:}A.b)(a) \simeq [a/x]b$)

$$\frac{\Gamma,\ x:A \vdash B\ type}{\Gamma \vdash \Pi x{:}A.B\ type} \quad \frac{\Gamma,\ x:A \vdash b:B}{\Gamma \vdash \lambda x{:}A.b : \Pi x{:}A.B} \quad \frac{\Gamma \vdash f : \Pi x{:}A.B \quad \overline{\Gamma} \vdash a : A}{\Gamma \vdash f(a) : [a/x]B}$$

Linear Π-*types* (Conversion: $(\lambda x{::}A.b)\ a \simeq [a/x]b$)

$$\frac{\Gamma,\ x{::}A \vdash B\ type}{\Gamma \vdash \Pi x{::}A.B\ type} \quad \frac{\Gamma,\ x{::}A \vdash b:B}{\Gamma \vdash \lambda x{::}A.b : \Pi x{::}A.B} \quad \frac{\Gamma \vdash f : \Pi x{::}A.B \quad \Delta \vdash a : A \quad Merge(\Gamma;\Delta)\downarrow}{Merge(\Gamma;\Delta) \vdash f\ a : [a/x]B}$$

Figure 1: Intuitionistic and linear Π-types

Equality types (Conversion: $subst(refl(a), q) \simeq q$)

$$\frac{\Gamma \vdash a : A \quad \Delta \vdash b : A \quad merge(\Gamma;\Delta)\downarrow}{merge(\Gamma;\Delta) \vdash Eq_A(a,b)\ type} \quad \frac{\Gamma \vdash a : A}{\Gamma \vdash refl(a) : Eq_A(a,a)}$$

$$\frac{\Gamma \vdash p : Eq_A(a,b) \quad \Delta \vdash q : B[a] \quad Merge(\Gamma;\Delta), x{:}^{\bar{:}}A \vdash B[x]\ type\ (^{\bar{:}} \in \{:,::\}) \quad Merge(\Gamma;\Delta)\downarrow}{Merge(\Gamma;\Delta) \vdash subst(x.B,p,q) : B[b]}$$

Figure 2: Equality types

weak linearity. More precisely, for $\Gamma \vdash a : A$, the multiset of variables essentially occurring in a under Γ, $E_\Gamma(a)$, is defined by induction on derivations (we omit the part of the definition for Eq-types): (1) for (V) above, $E_{\Gamma,\ x{:}^{\bar{:}}A,\ \Gamma'}(x) = D_{\Gamma,\ x{:}^{\bar{:}}A,\ \Gamma'}(x)$; (2) for the λ-typing rules, $E_\Gamma(\lambda x{:}^{\bar{:}}A.b) = E_{\Gamma,x{:}^{\bar{:}}A}(b)\backslash\{x\}$, where $^{\bar{:}} \in \{:,::\}$; (3) for intuitionistic applications, $E_\Gamma(f(a)) = E_\Gamma(f) \cup E_{\overline{\Gamma}}(a)$; (4) for linear applications, $E_{Merge(\Gamma;\Delta)}(f\ a) = E_\Gamma(f) \cup E_\Delta(a)$.

Theorem (weak linearity) If $\Gamma \vdash a : A$ and $x{::}\Gamma_x \in \Gamma$, then $x \in E_\Gamma(a)$ only once. □

Implementation and future work. We have implemented a prototype of the type checking algorithm for LDTT, which includes the rules in Figures 1 and 2. The code can be found at `https://github.com/yveszhang/ldtyping`.

LDTT as presented shows a way to introduce types that may depend on linear variables. Future work includes the study of extensions to other type constructors. Dependent types were introduced into the Lambek calculus in [4] and it would be interesting to see how our work above can be incorporated to allow type dependency on Lambek variables.

References

[1] I. Cervesato and F. Pfenning. A linear logical framework. *Information and Computation*, 179, 2002.

[2] J.-Y. Girard. Linear logic. *Theoret. Comput. Sci.*, 50, 1987.

[3] N. Krishnaswami, P. Pradic, and N. Benton. Integrating dependent and linear types. *POPL 2015*.

[4] Z. Luo. A Lambek Calculus with Dependent Types. *TYPES 2015*.

[5] M. Vákár. A categorical semantics for linear logical frameworks. *FoSSaCS 2015*.

[6] D. Walker. Substructural type systems. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–43. MIT, 2005.

On Unification of Lambda Terms

Giulio Manzonetto[1]
Andrew Polonsky[2]

[1] Université Paris 13, Laboratoire LIPN, CNRS UMR 7030, France
[2] Université Paris Diderot, Laboratoire IRIF, CNRS UMR 8243, France

We propose a unification algorithm for the type-free lambda calculus. As an intermediate step in this development, we are led to the following question:

What does it mean to unify untyped higher-order terms?

As an illustrative example, consider the obsevation, due to Visser, that Ω is the only λ-term which reduces to itself in a single step (up to a redex-free context around it). Wlog, we may assume that the redex is at the root; then any term M with this property must be a solution to a "unification equation"

$$M = (\lambda v.X)Y = X[Y/v] \tag{1}$$

where X,Y are meta-variables for terms.

(Here and henceforth, "=" denotes syntactic α-equality.)

Note that $X = X(v)$ is a higher-order meta-variable — but it is not a "context", because it can have more than one hole.

We may pursue to solve (1) analogously to the usual unification process. Since the substitution result $X[Y/v]$ is an application, we have two possibilities for $X = X(v)$:

$$X(v) = \begin{cases} v & (A) \\ X_1(v)X_2(v) & (B) \end{cases}$$

Case (A), together with (1), yields $(\lambda v.X)Y = X(Y) = Y$, contradicting finiteness of M. (Although, the infinite term $Y = (\lambda v.v)Y$ *is* a solution of (1), and it is indeed a 1-cycle in the infinitary lambda calculus.)

So (A) is ruled out. Rewriting (1) using (B), we obtain

$$(\lambda v.X_1(v)X_2(v))Y = (\lambda v.X)Y = X[Y/v] = X_1(Y)X_2(Y)$$
$$X_1(Y) = \lambda v.X_1(v)X_2(v) \tag{2}$$
$$X_2(Y) = Y \tag{3}$$

Next, we can "destruct" X_1 according to (2), obtaining two possibilities ($X_1(v) = v$ and $X_1(v) = \lambda v'.X_{10}(v, v')$), eliminate one, rewrite the other back

into (2), and so forth. Proceeding in this manner, we shall eventually find that the only solution is $M = (\lambda x.xx)(\lambda x.xx)$. (See [2] for other examples.)

In a similar manner, we can enumerate all terms with a specified "subterm pattern", given by a set of unification constraints. Such an algorithm could be used to enumerate all terms with a particular redex structure (for example, in order to analyze possible standardard reductions of shape $Nx \twoheadrightarrow x(Nx)$).

It may not be obvious that the above process should always terminate — unlike the case of first-order unification, substituting a meta-variable does not completely eliminate it. Each iteration develops the root symbol only, and may yet introduce new meta-variables, occurring deeper and more often (while substitution proliferates and rearranges existing ones).

Nevertheless, we prove that the above algorithm does terminate, and yields a set of most general solutions. (There are in general many of them.)

At first, our claim appears to contradict the common wisdom that "higher-order unification is undecidable" [1]. Doesn't it?

First of all, notice that we are working in an untyped setting, whereas higher-order unification is usually considered for simple types. While this does not seem to help much, it may perhaps explain the second difference, which is that *we do not contract the beta redexes*.

Indeed, from the perspective of equational logic, unification is a purely syntactic operation, and is not expected to take into consideration the whole set of axioms a given theory might have.

The lambda calculus is a theory of terms built up with a binary first-order operation and a unary higher-order operation; the beta-rule is merely an axiom of this theory. Generalizing to other higher-order signatures, with perhaps different rewrite rules, one could scarcely hope the generic "unification algorithm" to subsume all their possible equations.

So we do not solve full second-order unification. At the same time, the higher-order meta variables are not inert, either. In a sense, the substitutions being performed at every step are morally contractions of beta-redexes introduced by the algorithm implicitly.

Finally, this approach does not seem to fall under context unification either, since the occurrence of "holes" can be multiple and unbounded.

In the original spirit of the Types meetings, we hope that our presentation may elicit suggestions from the audience toward how our result relates to the well-known unification paradigms.

References

[1] G. Dowek, *Higher-Order Unification and Matching*, Handbook of Automated Reasoning, pp. 1009–1062, Elsevier and MIT Press, 2001

[2] M. Venturini Zilli, *Reduction Graphs in the Lambda Calculus*, Theor. Comput. Sci., Vol. 29, pp. 251–275, 1984.

A Structural Approach to the Stretching Lemma of Simply-Typed Lambda-Calculus

Ralph Matthes*

Institut de Recherche en Informatique de Toulouse (IRIT),
C.N.R.S. and University of Toulouse, France

This work is about an analysis of an advanced aspect of the inhabitation problem in simply-typed λ-calculus by way of a λ-calculus notation for proof search in minimal implicational logic, introduced in joint work with José Espírito Santo and Luís Pinto [1] (see also the revised and extended version [2]). 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 [5] goes into the opposite direction and considers λ-terms where there is only one term variable per type. In the joint work of the author, 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 in his 1979 PhD thesis with a very concrete λ-calculus approach and by Takahashi et al [5] 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 [1, 2]:

$$\begin{array}{lrcl} \text{(terms)} & N & ::= & \lambda x^A.N \mid \mathsf{gfp}\ X^\sigma.E_1 + \cdots + E_n \mid X^\sigma \\ \text{(elimination alternatives)} & E & ::= & xN_1 \dots N_k \end{array}$$

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 are given a semantics. 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 (modulo a notion of bisimilarity that considers the sums of alternatives as sets) $\mathcal{S}(\sigma)$ [1].

In subsequent work, based on this framework, the same authors studied (among other questions) a decision algorithm for the problem "given a type A, is there only a finite number of β-normal η-long inhabitants of type A in simply-typed λ-calculus?" [3] (with only a few more references, unfortunately missing out the work by Zaionc, e.g., with David, and with comments on PSPACE-completeness). While in that work, the only quantitative information

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

that is studied concerns the number of such inhabitants (which would often be smaller under the total discharge convention), the present work refines the analysis of the decision procedure with quantitative information on the depth (height ignoring λ-abstractions, as defined in Hindley's book [4], but a similar notion where variables are assigned value 1 instead of 0 is called height—of Böhm trees—by Takahashi et al [5]).

A measure δ on the terms involving fixpoint variables is defined analogously to depth and height, by ignoring λ-abstractions and by maximizing over sums of elimination alternatives and over arguments (even as in the height definition), but an application is reduced to 0 if any argument has measure 0. And a fixpoint variable counts ∞. One can show that for any term T, $\delta(T) = 0$ iff a decidable predicate of T holds that, for closed terms, is known to characterize the absence of inhabitants [3]. Another result of that previous work characterizes by a predicate FF the closed terms that have only finitely many inhabitants (to be more precise, the characterizations, applied to terms of the form $\mathcal{F}(\sigma)$, have this form, but they need to specify the general situation of not necessarily closed T). The latter result is refined in the present work by bounding the depth of all inhabitants to strictly below $\delta(\mathcal{F}(\sigma))$ in case $\mathsf{FF}(\mathcal{F}(\sigma))$—again, a statement for all T had to be found. This result is always informative since—as can be shown—for all closed T, $\mathsf{FF}(T)$ implies $\delta(T) < \infty$.

Ben-Yelles proved the stretching lemma (see Hindley's book [4, 8D2]) saying that if there is an η-long β-normal inhabitant of type A of depth at least the number n of atoms occurring in A, then there are infinitely many such inhabitants. Turning it around, if there are only finitely many inhabitants, then all inhabitants have depth strictly less than n. By the result above, an alternative proof of the stretching lemma would be obtained by showing that $\delta(\mathcal{F}(\sigma)) \leq n$, with σ the sequent that asks for A in the empty context, under the proviso $\mathsf{FF}(\mathcal{F}(\sigma))$.

Currently, only the case $n = 1$—called the monatomic case [4]—has been solved by the author in this manner. Apart from the trivial cases of an isolated atom (no inhabitant) or an implication without any nesting ($\delta = 1$), at least one composite hypothesis appears. In this case, $\mathsf{FF}(\mathcal{F}(\sigma))$ implies $\delta(\mathcal{F}(\sigma)) = 0$. The proof needs to be more informative in refuting the condition in case any of the appearing hypotheses is the atom alone. And it needs to be more general in speaking also about terms appearing in the construction process for $\mathcal{F}(\sigma)$; it then goes by induction on the term structure (with provisos since the statement is even false for fixpoint variables alone).

The advantage of the proposed method (as is the case already of the cited work it is refining) is that most arguments are by recursion on structure, in particular on the expressions of the λ-calculus for proof search. Hopefully, the full stretching lemma will be obtained following the same path.

References

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

[2] José Espírito Santo, Ralph Matthes, and Luís Pinto. A calculus for a coinductive analysis of proof search. http://arxiv.org/abs/1602.04382, February 2016.

[3] 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/1604.02086, April 2016.

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

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

Toward a computational reduction of dependent choice in classical logic to system F

Étienne Miquey[1,2] and Hugo Herbelin[1]

[1] PiR2, INRIA, Institut de Recherche en Informatique Fondamentale, Université Paris-Diderot
hugo.herbelin@inria.fr, emiquey@pps.univ-paris-diderot.fr
[2] IMERL, Universidad de la República, Montevideo

The dependent sum type of Martin-Löf's type theory provides a strong existential elimination, which allows to prove the full axiom of choice. The proof is simple and constructive:

$$\begin{array}{rcl} AC_A & := & \lambda H.(\lambda x.\,\mathtt{wit}(Hx), \lambda x.\,\mathtt{prf}(Hx)) \\ & : & \forall x^A \exists y^B P(x,y) \to \exists f^{A\to B} \forall x^A P(x, f(x)) \end{array}$$

where `wit` and `prf` are the first and second projections of a strong existential quantifier.

We present here a proof system which provides a proof-as-program interpretation of classical arithmetic with dependent choice, together with a computational reduction of this calculus to an intuitionistic one by means of a continuation-and-state-passing style translation. This system is a sequent-calculus version of Herbelin's dPA^ω calculus [5], who proposed a way of scaling up Martin-Löf proof to classical logic. The main ideas are first to restrict the dependent sum type to a fragment of the calculus to make it computationally compatible with classical logic, second to represent a countable universal quantification as an infinite conjunction. This allows to internalize into a formal system the realizability approach [2, 4] as a direct proof-as-programs interpretation.

Informally, let us imagine that given $H : \forall x^A \exists y^B P(x,y)$, we have the ability of creating an infinite term $H_\infty = (H0, H1, \ldots, Hn, \ldots)$ and select its n^{th}-element with some function `nth`. Then one might wish that

$$\lambda H.(\lambda n.\,\mathtt{wit}(\mathtt{nth}\ n\ H_\infty), \lambda n.\,\mathtt{prf}(\mathtt{nth}\ n\ H_\infty))$$

could stand for a proof for $AC_{\mathbb{N}}$. However, even if we were effectively able to build such a term, H_∞ might contain some classical proof. Therefore two copies of H_n might end up being different according to their context in which they are executed, and then return two different witnesses. This problem could be fixed by using a shared version of H_∞, say

$$\lambda H.\,\mathtt{let}\ a = H_\infty\ \mathtt{in}\ (\lambda n.\,\mathtt{wit}(\mathtt{nth}\ n\ a), \lambda n.\,\mathtt{prf}(\mathtt{nth}\ n\ a))\,.$$

It only remains to formalize the intuition of H_∞. We do this by a stream $\mathtt{cofix}^0_{fn}(Hn, f(S(n)))$ iterated on f with parameter n, starting with 0 :

$$\begin{array}{rl} AC_{\mathbb{N}} := \lambda H.\,\mathtt{let}\ a = \mathtt{cofix}^0_{fn}\ (Hn, f(S(n))\ \mathtt{in} & \\ (\lambda n.\,\mathtt{wit}(\mathtt{nth}\ n\ a), \lambda n.\,\mathtt{prf}(\mathtt{nth}\ n\ a))\,. & \end{array}$$

Whereas the stream is, at level of formulæ, an inhabitant of a coinductively defined infinite conjunction $\nu^0_{Xn}(\exists P(0,y) \wedge X(n+1))$, we cannot afford to pre-evaluate each of its components, and thus have to use a *lazy* call-by-value evaluation discipline. However, it still might be responsible for some non-terminating reductions.

We intend to tackle the problem by progressively reducing the consistency of our system to the normalization of Girard-Reynold's system F. However, the sharing forces us to design a state-passing

style translation, whose small-step behaviour is quite far from the sharing strategy in natural deduction. Besides, in order to get a proof of normalization through such a translation, we also need to guarantee some typing properties in the source language and along the translation.

We presented a preliminary version of this work at TYPES 2015, where, as a first step, we managed to develop a sequent-calculus version of dPA^{ω}, adapting the call-by-need version of the $\bar{\lambda}\mu\tilde{\mu}$-calculus designed by Ariola et al. [1]. Incidentally, we had to ensure its compatibility with dependent types, since the $\bar{\lambda}\mu\tilde{\mu}$-calculus [3] does not allow it directly. This led us to a type system annotated with a dependencies list, and made us add delimited continuations to our language. Indeed, if we consider the case of a proof $\lambda a.p : [a : A] \to B$ cut with a context $q \cdot e$ where $q : A$ and $e : B[q]^{\perp\!\!\!\perp}$, it usually reduces to the command $\langle q \mid \tilde{\mu}a.\langle p \mid e\rangle\rangle$ where $p : B[a]$ and $e : B[q]^{\perp\!\!\!\perp}$ are of incompatible types. While a annotation (to link a and q) on the type system can solve this, there is no hope that a direct continuation-passing style translation could be well-typed. Thus we introduced delimited continuations to turn it into a command $\langle \mu\hat{\mathfrak{p}}.\langle q \mid \tilde{\mu}a.\langle p \mid \hat{\mathfrak{p}}\rangle\rangle \mid e\rangle$ where p will not be cut with e until a is replaced by q.

The work is still in progress and in this talk, we propose to focus on the second step, that is the design of a continuation-and-state-passing style translation that is correct with respect to types and computation. As in [1], we benefited from Danvy's methodology of semantic artifacts. We first derive a small-step reduction system, to obtain a context-free abstract machine in which at each step a decision over a command $\langle p \mid e\rangle$ can be made by examining either the proof p or the context e in isolation. To do so, we separate the reductions rule in two different layers, which intuitively correspond to the call-by-value and store-management for the first one, and to the core computations for the second one.

This small-step system almost gives us directly a state-passing style translation. The remaining difficulty is to type the store in the target language, which is a quite subtle problem due to the fact that the store can be expanded in a non-linear way when unfolding a `cofix`. It is our hope that we could use the second-order quantification of system F to encode the store and its expansion, which would provide us with a proof of equiconsistency between classical arithmetic with dependent choice and system F.

Surprisingly, it turns out that our construction does not require any use of dependent choice at the meta-level. If some previous works [2, 6] succeeded in giving a computational content to the axioms of dependent choice or bar induction, this is to the best of our knowledge the first one that does not need any meta-use of one of these axioms.

References

[1] Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis Saurin, *Classical call-by-need sequent calculi: The unity of semantic artifacts*, FLOPS 2012, Proceedings, 2012, pp. 32–46.

[2] Stefano Berardi, Marc Bezem, and Thierry Coquand, *On the computational content of the axiom of choice*, J. Symb. Log. **63** (1998), no. 2, 600–622.

[3] Pierre-Louis Curien and Hugo Herbelin, *The duality of computation*, ICFP, 2000, pp. 233–243.

[4] Martín H. Escardó and Paulo Oliva, *Bar recursion and products of selection functions*, CoRR **abs/1407.7046** (2014).

[5] Hugo Herbelin, *A constructive proof of dependent choice, compatible with classical logic*, Logic in Computer Science, LICS 2012, Proceedings, IEEE Computer Society, 2012, pp. 365–374.

[6] J.-L. Krivine, *Dependent choice, 'quote' and the clock*, Th. Comp. Sc. **308** (2003), 259–276.

A Guide to the Mizar Soft Type System*

Adam Naumowicz and Josef Urban

[1] University of Bialystok, Poland
[2] Czech Technical University in Prague, Czech Republic

Introduction Mizar [1, 3] is in a way both typed and untyped. In a foundational sense, Mizar is based on untyped set theory. The set-theoretical world initially consists of many objects of "just one type". However, the objects can have various properties (a number, ordinal number, complex number, Conway number, a relation, function, complex function, complex matrix), however none of them is considered to be of "foundational importance", and all these properties are treated as equal "adjectives" or "attributes", which are semantically just (dependent) predicates.

Typically, mathematicians are interested only in some particular properties of the objects, and fluently shift and focus between them, and take some of them as granted in various contexts. Thus, the set of natural numbers can once be treated as a measurable cardinal, another time as a subset of real (and thus "obviously" complex) numbers. A group can simultaneously be a subgroup of other groups. The Mizar "typing" mechanisms are concerned with providing such fluid and "obvious" treatment of the concepts, once the particular relations have been established *in the underlying logic*. This is especially important in large mathematical theories, where complicated and unpredictable hierarchies (ontologies) between the defined concepts arise, and it is often impossible to pre-design a "perfect type system" between the objects, that would specify upfront all the disjointness, inclusion, and other relations. An important part of the Mizar typing mechanisms is thus dynamic type change.

There have been numerous attempts to reconstruct elements of this type system in order to translate the mathematical data encoded in the Mizar language into other common mathematical data exchange formats like OMDoc [4], other proof assistants like HOL Light [6] or Isabelle [5]. The soft-typing mechanisms of Mizar have been actively used in MML already before 1990, i.e., before languages like Haskell started to be developed and provided motivation for the recent soft-typing ("type-class") mechanisms in systems like Coq and Isabelle. A particular advantage of the soft-typing approach is its straightforward translation to first-order ATP formats. This enabled the first encouraging ATP experiments over the whole Mizar library [8], leading to the expansion of hammer-style ITP methods in the last decade.

Types, modes, attributes and adjectives When any variable is introduced in Mizar, its **type** must be given. And for any term, the verifier computes its unique type. Types are then used in quantified and qualifying formulas, for parsing, semantic analysis, overloading resolution, and inferring object properties. Simple types in Mizar are constructed using **modes** and the constructors of **adjectives** are called **attributes**. Mizar supports two kinds of mode definitions: modes defined as a collection (called a cluster) of adjectives associated with an already defined radix type to which they may be applied, called expandable modes, and modes that define a type with an explicit definiens that must be fulfilled for an object to have that type. One of the features of the Mizar type system is that the types must be non-empty, i.e. there must exist at least one object of a given type. This restriction is introduced to guarantee that the formalized theory always has some denotation. Mode definitions thus require a proof of that fact stated in the form of the `existence` condition. The mother type in a mode definition is used to specify the direct predecessor of the defined mode in the tree of Mizar types (representing the type widening relation). The type constructed with the new mode widens to its mother type. The widening relation also takes into account the adjectives that come with types, i.e. the type with a shorter list of (comparable) adjectives is considered to be wider. Mizar types

*The authors are supported by COST Action CA15123. J. Urban was funded by ERC project 649043 – AI4REASON.

are **dependent**. They can have an empty list of arguments, but most commonly they have explicit and/or implicit arguments. Adjectives can also be expressed with their own visible arguments, e.g., `n-dimensional`, or `X-valued`.

Type Change Mechanisms Types of mathematical objects defined in the Mizar library form a sup-semilattice with widening (subtyping) relation as the order [2]. There are two hierarchies of types: the main one based on the type `set`, and the other based on the notion of structure. The most general type in Mizar (to which both sets and structures widen) is called `object`. Structures in Mizar can be used to model mathematical notions like groups, topological spaces, categories, etc. which are usually represented as tuples. Mizar supports multiple inheritance of structures that makes a whole hierarchy of interrelated structures available in the Mizar library, with the `1-sorted` structure being the common ancestor of almost all other structures. An important extension of the Mizar structure system is the possibility to define structures parameterized by arbitrary sets, or other structures.

The effective (semantic) type of a given Mizar term is determined by a number of factors - most importantly, by the available (imported from the library or introduced earlier in the same formalization) redefinitions and adjective registrations. **Redefinitions** are used to change the definiens or type for some constructor if such a change is provable with possibly more specific arguments. Depending on the kind of the redefined constructor and the redefined part, each redefinition induces a corresponding correctness condition that guarantees that the new definition is compatible with the old one. In the Mizar language, the common name **registration** refers to several kinds of Mizar features connected with automatic processing of the type information based on adjectives [7]. Grouping adjectives in so called clusters (hence the keyword cluster used in their syntax) enables automation of some type inference rules. Existential registrations are used to secure the nonemptiness of Mizar types. The dependencies of adjectives recorded as conditional registrations are used automatically by the Mizar verifier.

Sometimes, for syntactic (identification) purposes, e.g. to force the system use one of a number of matching redefinitions, the type of a term can be explicitely qualified to one which is less specific, e.g. `1 qua real number` whereas in standard environments the constant has the type `natural number` and then appropriate (more specific) definitions apply to it. The `reconsider` statement forces the system to treat any given term as if its type was the one stated (with extra justification provided), e.g. `reconsider R as Field` whereas the actual type of the variable `R` might be `Ring`. It is usually used if a particular type is required by some construct (e.g. definitional expansion) and the fact that a term has this type requires extra reasoning after the term is introduced in a proof.

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., Four Decades of Mizar - Foreword. Journal of Automated Reasoning 55(3), pp. 191-198, 2015

[4] Iancu, M., Kohlhase, M., Rabe, F., Urban, J., The Mizar Mathematical Library in OMDoc: Translation and Applications, 191-202, Journal of Automated Reasoning (50:2), Springer 2013.

[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] J. Urban. MPTP - Motivation, Implementation, First Experiments. *Journal of Automated Reasoning*, 33(3-4):319–339, 2004.

β reduction without rule ξ

Masahiko Sato and Randy Pollack

It is well known that, for β reduction of pure λ terms, the ξ rule is invertible:

$$\lambda x.s \xrightarrow{\beta} \lambda x.t \implies s \xrightarrow{\beta} t$$

With this observation we give a de Bruijn-like representation of pure λ terms, and rules for β reduction in this representation that need no rule ξ because rule ξ is admissible. This work has been formalized in Isabelle/HOL and proved adequate w.r.t. nominal Isabelle.

Fix a countable set of names, ranged over by x, y. Let i, j, m, n range over natural numbers. The raw syntax of preterms is

$$\mathsf{pt} ::= \mathsf{X}_n\, x \mid \mathsf{J}_n\, j \mid (M\ N)_n$$

Preterms are ranged over by M, N, P, Q, and indexed by their *height*, n (write $hgt\, M$). There is a notion of *well formedness* of preterms, $\mathcal{W}\, M$, defined inductively by

$$\frac{}{\mathcal{W}\, \mathsf{X}_n\, x} \qquad \frac{i < n}{\mathcal{W}\, \mathsf{J}_n\, i} \qquad \frac{\mathcal{W}\, M \quad \mathcal{W}\, N \quad n \le hgt\, M \quad n \le hgt\, N}{\mathcal{W}\, (M\ N)_n}$$

If $\mathcal{W}\, M$ we call M a *term*, and write $\mathcal{W}_n\, M$ to mean $\mathcal{W}\, M$ and $n \le hgt\, M$. The height of a term shows how many bindings it implicitly sits under.

We can define *abstraction* as a function on preterms:

$$\begin{aligned} \mathsf{lam}_x(\mathsf{X}_n\, y) &:= \text{if } x = y \text{ then } \mathsf{J}_{n+1}\, 0 \text{ else } \mathsf{X}_{n+1}\, y \\ \mathsf{lam}_x(\mathsf{J}_n\, j) &:= \mathsf{J}_{n+1}\, (j{+}1) \\ \mathsf{lam}_x((M\ N)_n) &:= (\mathsf{lam}_x(M)\ \mathsf{lam}_x(N))_{n+1} \end{aligned}$$

Abstraction preserves well formedness and raises height by one.

$$\mathcal{W}_n\, M \implies \mathcal{W}_{n+1}\, \mathsf{lam}_x(M)$$

Conversely, every term with height a successor is an abstraction. We use A, B as metavariables over abstractions.

The intended interpretation of preterms is given by the relation

$$x \sim \mathsf{X}_0\, x \qquad \frac{t_1 \sim M_1 \quad t_2 \sim M_2}{(t_1\, t_2) \sim (M_1\ M_2)_0} \qquad \frac{t \sim M}{\lambda x.t \sim \mathsf{lam}_x(M)}$$

which is an isomorphism between conventional λ terms (e.g. nominal terms) and terms of our formal lamguage.

To define instantiation we first introduce a lifting function

$$(\mathsf{X}_n\, y)^{\uparrow} := \mathsf{X}_{n+1}\, y \qquad (\mathsf{J}_n\, j)^{\uparrow} := \mathsf{J}_{n+1}\, (j{+}1) \qquad ((M\ N)_n)^{\uparrow} := ((M)^{\uparrow}\ (N)^{\uparrow})_{n+1}$$

which we iterate as: $(M)^{\uparrow 0} := M$ and $(M)^{\uparrow m+1} := ((M)^{\uparrow m})^{\uparrow}$.

Instantiation is a binary function, $M[N]$. If $hgt\, M = 0$ (M is under no binders), $M[N] = M$. Otherwise $M[N]$ fills any holes $\mathsf{J}_{n+1}\, 0$ in M and adjusts the rest of the term:

$$\begin{aligned} \mathsf{X}_{n+1}\, y[N] &:= \mathsf{X}_n\, y \qquad & \mathsf{J}_{n+1}\, 0[N] &:= (N)^{\uparrow n} \qquad & (M\ P)_{n+1}[N] &:= (M[N]\ P[N])_n \\ & & \mathsf{J}_{n+1}\, (j{+}1)[N] &:= \mathsf{J}_n\, j & & \end{aligned}$$

Instantiation preserves well formedness and lowers height by one:

$$\mathcal{W}_{n+1}\, M \wedge \mathcal{W}\, N \implies \mathcal{W}_n\, M[N]$$

Using abstraction we have a natural definition of β reduction:

$$\frac{\mathcal{W}\, M \quad \mathcal{W}\, N}{(\mathsf{lam}_x(M)\ N)_0 \xrightarrow{\beta} (\mathsf{lam}_x(M))[N]}$$

$$\frac{M \xrightarrow{\beta} M' \quad \mathcal{W}\, N}{(M\ N)_0 \xrightarrow{\beta} (M'\ N)_0} \qquad \frac{\mathcal{W}\, M \quad N \xrightarrow{\beta} N'}{(M\ N)_0 \xrightarrow{\beta} (M\ N')_0} \qquad \frac{M \xrightarrow{\beta} N}{\mathsf{lam}_x(M) \xrightarrow{\beta} \mathsf{lam}_x(N)}$$

Any preterm that participates in this relation is well-formed. This relation is correct β reduction w.r.t. the meaning of preterms given above, but still contains an invertible ξ rule. To define an equivalent relation with no ξ rule we need to define *generalized lifting*, $(M)^{i\Uparrow}$:

$$(\mathsf{X}_n\, y)^{i\Uparrow} := \mathsf{X}_{n+1}\, y \qquad (\mathsf{J}_n\, j)^{i\Uparrow} := \begin{cases} \mathsf{J}_{n+1}\, j & (j < i) \\ \mathsf{J}_{n+1}\, (j{+}1) & (j \geq i) \end{cases} \qquad ((M\ N)_n)^{i\Uparrow} := ((M)^{i\Uparrow}\ (N)^{i\Uparrow})_{n+1}$$

which we iterate as $(M)^{i\Uparrow 0} := M$ and $(M)^{i\Uparrow m+1} := ((M)^{i\Uparrow m})^{i\Uparrow}$. As with instantiation, *generalized instantiation*, $(M)[N]^i$, leaves terms M of height 0 unchanged, and updates abstractions:

$$(\mathsf{X}_{n+1}\, y)[M]^i := \mathsf{X}_n\, y \quad (\mathsf{J}_{n+1}\, i)[M]^i := (M)^{i\Uparrow n-i} \qquad ((P\ Q)_{n+1})[M]^i := ((P)[M]^i\ (Q)[M]^i)_n$$

$$(\mathsf{J}_{n+1}\, j)[M]^i := \begin{cases} \mathsf{J}_n\, j & (j < i) \\ \mathsf{J}_n\, (j{-}1) & (j > i) \end{cases}$$

Claim the relation $\bullet > \bullet$ defined without a ξ rule:

$$\frac{\mathcal{W}_{n+1}\, A \quad \mathcal{W}_n\, N}{(A\ N)_n > (A)[N]^n} \qquad \frac{M > M' \quad \mathcal{W}_n\, M \quad \mathcal{W}_n\, N}{(M\ N)_n > (M'\ N)_n} \qquad \frac{N > N' \quad \mathcal{W}_n\, M \quad \mathcal{W}_n\, N}{(M\ N)_n > (M\ N')_n}$$

is equivalent to the relation $\bullet \xrightarrow{\beta}_{\beta} \bullet$ given above (and thus to the usual notion of β reduction).
Proof that $M > N \implies M \xrightarrow{\beta} N$ goes by induction on the relation $M > N$. Both congruence rule cases use invertibility of rule ξ for the relation $\bullet \xrightarrow{\beta} \bullet$. The converse direction is straightforward. □

Here is Tait–Martin-Löf parallel reduction without a ξ rule.

$$\frac{}{\mathsf{X}_n\, y \gg \mathsf{X}_n\, y} \qquad \frac{n \leq hgt\, M \quad M \gg M' \quad n \leq hgt\, N \quad N \gg N'}{(M\ N)_n \gg (M'\ N')_n} \qquad \frac{j < n}{\mathsf{J}_n\, j \gg \mathsf{J}_n\, j}$$

$$\frac{n < hgt\, A \quad A \gg B \quad n \leq hgt\, M \quad M \gg N}{(A\ M)_n \gg (B)[N]^n}$$

This (nondeterministic) parallel reduction can be made into (deterministic) complete development by replacing the application congruence rule with

$$\frac{n = hgt\, M \quad M \gg M' \quad n \leq hgt\, N \quad N \gg N'}{(M\ N)_n \gg (M'\ N')_n}$$

which removes overlap with the β rule.

Unfortunately this approach doesn't seem to extend to $\beta\eta$ reduction, as rule ξ is not invertible in that case. On this point it is interesting to note that none of the reduction relations in this note can reduce the height of a term, but η reduction can do that.

The Dialectica Translation of Type Theory

Andrej Bauer[1] and Pierre-Marie Pédrot[2]

[1] University of Ljubljana
[2] INRIA

The Dialectica translation, introduced by Gödel in the eponymous journal [2] is a logical translation from intuitionistic higher-order arithmetic $\mathbf{HA}^\omega$ into System $\mathbf{T}$, a simply-typed λ-calculus with integers. In modern terms, one would call it a *realizability* interpretation, as all logical content is shifted into the meta-theory: proofs are erased into simply-typed realizers while logic is interpreted thanks to an orthogonality relation, allowing to discriminate pseudo-proofs from actual proofs. Thanks to this interpretation, one can realize two additional principles which are not provable in $\mathbf{HA}^\omega$, namely Markov's principle and the independence of premise.

Contrarily to Kreisel's modified realizability, which actually originated as a variant of Dialectica, realizers produced by the latter are not a mere computational erasure of the corresponding proof-term. Just as the Lafont-Reus-Streicher (LRS) CPS translation [3] can be synthetized back from Krivine's realizability [5], there is a novel program translation hidden beneath Dialectica's realizability. In a previous paper [6], the second author gave a dynamic explanation of an instance of this translation on the simply-typed λ-calculus by means of the Krivine machine, which showed that Dialectica gives a first-class account of two notions coming from the machine: 1. stacks, which are given a proper *counter* type $\mathbb{C}(A)$ for any source type A and 2. substitution, which is present in the machine in the form of closures. While stacks are already present in the LRS translation, first-class citizenship of substitution is not usually present in program translations, except maybe for call-by-need encodings.

In a nutshell, the simply-typed translation goes as follows. To any type A, it associates a witness type $\mathbb{W}(A)$, a counter type $\mathbb{C}(A)$ and an orthogonality relation $\perp_A$. To a term $\Gamma \vdash t : A$, it associates a term $\mathbb{W}(\Gamma) \vdash t^\bullet : \mathbb{W}(A)$ together with a family of terms $\mathbb{W}(\Gamma) \vdash t_x : \mathbb{C}(A) \to \mathfrak{M}\,\mathbb{C}(X)$ for each $(x : X) \in \Gamma$, where $\mathfrak{M}\,(-)$ stands for the finite multiset monad. This translation is computationally sound, i.e. assuming reasonable commutation rules on the multiset monad, it preserves conversion. Its direct-style interpretation can be thought of as a mix of an instrumentation of variable accesses together with a weak form of delimited continuations. Morally, t_x does the following: execute t in the machine, and each time x is dereferenced, add the current continuation (i.e. the stack of the machine) to a global multiset which is finally returned. This effect is commutative, in so far as stacks are returned without regard of the relative order of dereferencing, as we produce a multiset rather than a list. Such a desequentialization is vital for the preservation of conversion.

In the same paper, the second author showed that this translation readily adapts to the dependently-typed case, by providing a Dialectica translation of the whole calculus of construction with universes $\mathbf{CC}_\omega$ into $\mathbf{CC}_\omega$ plus Σ-types and a computational multiset monad. This shows that Dialectica is in a certain sense more intuitionistic than LRS, which is not known to exist in a dependent setting.

While Dialectica could also be independently adapted to handle simply-typed algebraic datatypes [7], the interpretation of dependent elimination remained difficult to tackle at first sight. Nonetheless, we show that by a simple tweak inspired by a call-by-push-value decomposition, one can recover the full dependent elimination principles, hence demonstrating that the Dialectica translation was indeed genuinely intuitionistic. The main theorem is the following.

Theorem 1. *Assuming reasonable commutation rules on the multiset monad, there exists a Dialectica translation from* **CIC** *into* $\mathbf{CIC} + \mathfrak{M}(-)$ *preserving typing and definitional equality.*

The main difference from the $\mathbf{CC}_\omega$ translation comes from the fact we have to make the counter type dependent on a witness to make the translation go through. It means that to any $A : \square$, we associate $\mathbb{W}(A) : \square$ and $\mathbb{C}(A) : \mathbb{W}(A) \to \square$ where $\mathbb{C}(A)$ is indexed. This triggers a handful of adaptations. A term $\Gamma \vdash M : A$ is now translated as:

1. A term $\mathbb{W}(\Gamma) \vdash M^\bullet : \mathbb{W}(A)$ (as before);
2. A family of terms $\mathbb{W}(\Gamma) \vdash M_x : \mathbb{C}(A)[M^\bullet] \to \mathfrak{M}\,\mathbb{C}(X)[x]$ for all $(x : X) \in \Gamma$.

Note how the type of counters produced by M_x depends on x, which is the crux of the trick. We give the translation of a few representative types to demonstrate how dependent the translation becomes.

	$\mathbb{W}(-) : \square$	$\mathbb{C}(-) : \mathbb{W}(-) \to \square$
$\square$	$\Sigma A^+ : \square . A^+ \to \square$	$\lambda\_ . 1$
$\Pi x : A . B$	$\Pi x : \mathbb{W}(A) . \Sigma y : \mathbb{W}(B) . \mathbb{C}(B)[y] \to \mathfrak{M}\,\mathbb{C}(A)[x]$	$\lambda f . \Sigma x : \mathbb{W}(A) . \mathbb{C}(B)[\pi_1\,(f\,x)]$
$\Sigma x : A . B$	$\Sigma x : \mathbb{W}(A) . \mathbb{W}(B)$	$\lambda(x, y) . \mathbb{C}(A)[x] + \mathbb{C}(B)[y]$
$A + B$	$\mathbb{W}(A) + \mathbb{W}(B)$	$\lambda[\, x \Rightarrow \mathbb{C}(A)[x] \mid y \Rightarrow \mathbb{C}(B)[y] \,]$

This provides new hindsights on the De Paiva's linear decomposition of Dialectica [1]. Indeed, usually $\mathbb{C}(A + B)$ is defined as $\mathbb{C}(A) \times \mathbb{C}(B)$, but it turns out it is instead a pattern-matching over the corresponding witness. In a non-dependent setting, the two definitions agree, but here we need to be smarter.

The translation extends **CIC** with unexpected reasoning principles and can be seen as a way to expose intensional contents of the source theory. Most notably, it negates functional extensionality in a meaningful way: one can observe how a function uses its arguments, and in particular how many times, by looking at the multisets produced by its second component. This is very similar to the techniques developed in quantitative semantics [4], except that it gives access to the full expressive power of **CIC** and thus paves the way for the design of a type theory featuring an internal notion of complexity.

References

[1] V. de Paiva. A dialectica-like model of linear logic. In *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 1989.

[2] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[3] Y. Lafont, B. Reus, and T. Streicher. Continuations semantics or expressing implication by negation. Technical Report 9321, Ludwig-Maximilians-Universitat, Munchen, 1993.

[4] J. Laird, G. Manzonetto, G. McCusker, and M. Pagani. Weighted relational models of typed lambda-calculi. In *LICS 2013*, pages 301–310, 2013.

[5] A. Miquel. Relating classical realizability and negative translation for existential witness extraction. In *TLCA 2009*, pages 188–202, 2009.

[6] P.-M. Pédrot. A functional functional interpretation. In *CSL-LICS 2014*, pages 77:1–77:10, New York, NY, USA, 2014. ACM.

[7] P.-M. Pédrot. *A Materialist Dialectica*. PhD thesis, Univ. Paris VII, 2015.

The Definitional Side of the Forcing

G. Jaber[1], G. Lewertowski[1], P.-M. Pédrot[2], M. Sozeau[1], and N. Tabareau[2]

[1] IRIF - Université Paris Diderot / πr^2 - Inria
[2] Inria Rennes - Bretagne Atlantique

Forcing has been introduced by Cohen to prove the independence of the Continuum Hypothesis in set theory. The main idea is to build, from a model M, a new model M' for which validity is controlled by a poset of forcing conditions living in M. Later, categorical ideas have been used by Lawvere and Tierney [10] to recast this construction in terms of topos of presheaves. It naturally gives rise to a proof-relevant forcing working on categories of conditions rather than simply posets.

Recent years have seen a renewal of interest for forcing, driven by Krivine's classical realizability [7]. In this line of work, forcing is studied as a proof translation, and one seeks to understand its computational content [9, 2], through the Curry-Howard correspondence. Following these ideas, a forcing translation heavily based on the presheaf construction of Lawvere and Tierney was defined in [5] for the Calculus of Inductive Constructions (**CIC**). The main goal was to extend the logic behind Coq with new principles, while keeping its fundamental properties: soundness, canonicity and decidability of type-checking. This approach can be seen, following [1], as type-theoretic metaprogramming.

However, this technique suffers from coherence problems, which complicate greatly the translation. More precisely, the translation of two definitionally equal terms are not in general definitionally equal, but only propositionally equal. Rewriting terms must then be inserted inside the definition of the translation. If this is possible to perform, albeit tedious, when the forcing conditions form a poset, it becomes intractable when we want to define a forcing translation parameterized by a category of forcing conditions.

We propose a novel forcing translation for the Calculus of Constructions which avoids these coherence problems. Departing from the categorical intuitions of the presheaf construction, it takes its roots in a call-by-push-value [8] decomposition of the previous translation. Through this decomposition, the new translation is *call-by-name*, while the previous one one is *call-by-value*. This is easily seen in the translation of dependent products where the type $\Pi x : A.\, B$ is interpreted at level p as $\Pi q \leq p.\, \Pi x : [\![A]\!]_q.\, [\![B]\!]_q$ in call-by-value v.s. $\Pi x : (\Pi q \leq p.\, [\![A]\!]_q).\, [\![B]\!]_p$ in call-by-name. Here, the argument x is boxed under a quantification over a future condition and morphism, which corresponds to the fact that it will be evaluated only when needed.

Assuming the forcing category verifies categorical laws definitionally, we get the following main result.

Call-by-name forcing provides the first effectful translation of the Calculus of Constructions into itself which preserves definitional equality.

The requirement on the forcing category is actually not an issue, as we can make any category abide by these definitional laws thanks to a type-theoretic Yoneda embedding.

This translation extends to inductive types by exploiting storage operators [6], an old idea of Krivine to simulate call-by-value in call-by-name in the context of classical realizability, restricting the power of dependent elimination in presence of effects. The necessity of a restriction should not be surprising and was already present in a similar work by Herbelin [4]. This is done by ensuring that dependent pattern-matchings are typed with other pattern-matchings, e.g. for the Σ-type elimination:

$$\frac{\Gamma \vdash M : \Sigma x : A.\, B \qquad \Gamma, z : \Sigma x : A.\, B \vdash C : \square \qquad \Gamma, x : A, y : B \vdash N : C\{z := (x, y)\}}{\Gamma \vdash \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N : \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow C\{z := (x, y)\}}$$

The original **CIC** can be retrieved by adding an η-law on inductive types which is not preserved by the translation. Actually, the translation allows to build non-canonical inhabitants of inductive types and thus negates this η-law. These non-canonical inhabitants can embed effectful computations. Hence,

Call-by-name forcing provides the first version of **CIC** *with effects.*

The nice property of preservation of definitional equality is emphasized by the implementation of a Coq plugin[1] which works for any term of **CIC**, assuming it complies with the restricted typing rules.

By using forcing, we produced various results around homotopy type theory. We proved that functional extensionality is preserved in any forcing layer. We also showed that the negation of Voevodsky's univalence axiom is consistent with **CIC** plus functional extensionality. This statement could already be deduced for the existence of a set-based *proof-irrelevant* model [11], but we provided the first formalization of it, in a proof relevant setting, and by an easy use of the forcing plugin. Under an additional assumption of parametricity, we showed conversely that we get the preservation of the univalence axiom.

This is a first step towards the use of the category of cubes as the type of forcing conditions to give a computational content to the cubical type theory [3] of Coquand et al. in Coq, and in particular to the univalence axiom.

References

[1] T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[2] A. Brunel. *The Monitoring Power of Forcing Transformation*. PhD thesis, Univ. Paris Nord, 2014.

[3] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom, 2015. Preprint.

[4] H. Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *LICS*, pages 365–374. IEEE Computer Society, 2012.

[5] G. Jaber, N. Tabareau, and M. Sozeau. Extending Type Theory with Forcing. In *LICS 2012 : Logic In Computer Science*, pages 0–0, Dubrovnik, Croatia, June 2012.

[6] J. Krivine. Classical logic, storage operators and second-order lambda-calculus. *Ann. Pure Appl. Logic*, 68(1):53–78, 1994.

[7] J.-L. Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009.

[8] P. B. Levy. *Call-by-push-value*. PhD thesis, Queen Mary, University of London, 2001.

[9] A. Miquel. Forcing as a program transformation. In *LICS*, pages 197–206. IEEE Computer Society, 2011.

[10] M. Tierney. Sheaf theory and the continuum hypothesis. In *Toposes, algebraic geometry and logic*, pages 13–42. Springer, 1972.

[11] B. Werner. Sets in types, types in sets. In *Theoretical aspects of computer software*, pages 530–546. Springer, 1997.

[1] Available at `https://github.com/CoqHott/coq-forcing`.

A Dialectica-Like Approach to Tree Automata

Colin Riba*

We propose a fibred monoidal closed category of alternating tree automata, based on a Dialectica-like approach to the fibred game model of [15].

Alternating tree automata are equivalent in expressive power to the Monadic Second-Order Logic on infinite trees (MSO), which subsumes most of the logics used in verification (see e.g. [5]). Acceptance of an input tree t by an automaton $\mathcal{A}$ can be described by a two-player game, where the *Proponent* P (also called *Automaton* or ∃loïse) tries to force the execution of the automaton on a successful path, while its *Opponent* O (∀belard) tries to find a failing path. Then $\mathcal{A}$ accepts t iff P has a winning strategy in this game. Alternating automata are easily closed under complement, and together with the translation of alternating automata to non-deterministic ones (the *Simulation Theorem* [12]), this provides a convenient decomposition of the translation of MSO formulas to automata (see e.g. [5]), implying the decidability of MSO [14]. This work shows that to some extent this decomposition can be reflected in the decomposition of intuitionistic logic in linear logic [4].

The fibred symmetric monoidal closed structure allows to organize tree automata in a deduction system for a first-order multiplicative linear logic. Our model, which is based on game semantics, provides following [15] a realizability interpretation of this system: From a derivation of say $\mathcal{A} \multimap \mathcal{B}$, we can compute a strategy σ witnessing the inclusion $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ (where $\mathcal{L}(\mathcal{A})$ is the set of trees accepted by $\mathcal{A}$), in the sense that for any input tree t and any strategy τ witnessing that $t \in \mathcal{L}(\mathcal{A})$, the strategy $t^*(\sigma) \circ \tau$ witnesses that $t \in \mathcal{L}(\mathcal{B})$ (here t^* is the substitution functor induced by t).

We use Gödel's *Dialectica* interpretation (see e.g. [1, 11]) in two related ways. First, transitions of alternating tree automata can be seen (following e.g. [12]) as being valued in a free distributive lattice, hence as being given by expressions in a $\vee\wedge$-form. Then, Dialectica, seen as a constructive notion of prenex $\exists\forall$-formulas, provides the transition function of the internal linear implication of our notion of tree automata. Second, our notion of morphism (issued from [15]) is based on *zig-zag* strategies, which can be represented by a Dialectica-like category (see e.g. [3, 7, 6]) based on the topos of trees (see e.g. [2]). This allows to conveniently describe the dependencies of strategies on tree directions, and to get a very simple fibred structure thanks to a variation of simple fibrations based on comonoid indexing (see e.g. [8]).

When restricting to parity automata, the winning conditions of games of the form $\mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n \multimap \mathcal{B}$ are given by disjunctions of parity conditions (called *Rabin conditions*), and it is known that if P wins such a game, then he has a positional winning strategy [10, 9]. In this context, we show that a powerset operation translating an alternating automaton to an equivalent non-deterministic one satisfies the deduction rules of

*LIP, Université de Lyon, CNRS, École Normale Supérieure de Lyon, INRIA, Université Claude-Bernard Lyon 1, colin.riba@ens-lyon.fr, http://perso.ens-lyon.fr/colin.riba/

the '!' modality of linear logic. Unfortunately, positional strategies do not compose, but we still get a deduction system for intuitionistic linear logic, which in particular gives deduction for minimal intuitionistic predicate logic via the Girard translation. Using a suitable negative translation based on the '?' modality, we can interpret proofs of minimal classical logic, and also get a weak form of completeness of our realizers w.r.t. language inclusion, in the sense that if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, then from a regular O-strategy witnessing $\mathcal{L}(!\mathcal{A}) \cap \mathcal{L}(!(\mathcal{B}^\perp)) = \emptyset$ (provided by an algorithm solving regular games on finite graphs, see e.g. [13]), we can build a winning P-strategy on $!\mathcal{A} \multimap (!(\mathcal{B}^\perp))^\perp$.

Details can be found in the unpublished draft [16].

References

[1] J. Avigad and S. Feferman. Gödel's functional ("Dialectica") interpretation. In S. Buss, editor, *Handbook Proof Theory*. Elsevier, 1998.

[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. *LMCS*, 8(4), 2012.

[3] V. de Paiva. The Dialectica categories. Technical Report 213, University of Cambridge Computer Laboratory, January 1991.

[4] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[5] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.

[6] P. J. W. Hofstra. The dialectica monad and its cousins. In M. Makkai and B.T. Hart, editors, *Models, Logics, and Higher-dimensional Categories: A Tribute to the Work of Mihály Makkai*, CRM proceedings & lecture notes. American Mathematical Society, 2011.

[7] J. M. E. Hyland. Proof theory in the abstract. *APAL*, 114(1-3):43–78, 2002.

[8] J. M. E. Hyland and A. Schalk. Glueing and orthogonality for models of linear logic. *Theoretical Computer Science*, 294(1/2):183–231, 2003.

[9] K. Klarlund and D. Kozen. Rabin Measures. *Chicago J. Theor. Comput. Sci.*, 1995, 1995.

[10] N. Klarlund. Progress measures, immediate determinacy, and a subset construction for tree automata. *Annals of Pure and Applied Logic*, 69(2-3):243–268, 1994.

[11] U. Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics*. Springer Monographs in Mathematics. Springer, 2008.

[12] D. E. Muller and P. E Schupp. Simulating Alternating Tree Automata by Nondeterministic Automata: New Results and New Proofs of the Theorems of Rabin, McNaughton and Safra. *Theor. Comput. Sci.*, 141(1&2):69–107, 1995.

[13] D. Perrin and J.-É. Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Pure and Applied Mathematics. Elsevier, 2004.

[14] M. O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[15] C. Riba. Fibrations of tree automata. In *TLCA*, volume 38 of *LIPIcs*, pages 302–316. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[16] C. Riba. A Dialectica-Like Approach to Tree Automata. Available at `http://perso.ens-lyon.fr/colin.riba/papers/dialaut.pdf`, 2016.

FLABloM: Functional linear algebra with block matrices

Adam Sandberg Eriksson and Patrik Jansson

Chalmers University of Technology, Sweden
{saadam,patrikj}@chalmers.se

In [1] Bernardy & Jansson used a recursive block formulation of matrices to certify Variant's [4] parsing algorithm. Their matrix formulation was restricted to matrices of size $2^n \times 2^n$ and this work extends the matrix formulation to allow for all sizes of matrices and applies similar techniques to algorithms that can be described as transitive closures of semi-rings of matrices with inspiration from [2] and [3].

We define a hierarchy of ring structures as Agda records. A semi-near-ring for some type s needs an equivalence relation $\simeq_s$, a distinguished element 0_s and operations addition $+_s$ and multiplication $\cdot_s$. Our semi-near-ring requires that 0_s and $+_s$ form a commutative monoid (i.e. $+_s$ commutes and 0_s is the left and right identity of $+_s$), 0_s is the left and right zero of $\cdot_s$, $+_s$ is idempotent ($\forall\ x \to x\ +_s\ x\ \simeq_s\ x$) and $\cdot_s$ distributes over $+_s$.

For the semi-ring we extend the semi-near-ring with another distinguished element 1_s and proofs that $\cdot_s$ is associative and that 1_s is the left and right identity of $\cdot_s$.

Finally we extend the semi-ring with an operation *closure* that computes the transitive closure of an element of the semi-ring (c is the closure of w if $c\ \simeq_s\ 1_s\ +_s\ w\ \cdot_s\ c$ holds), we denote the closure of w with w^*.

We use two examples of semi-rings with transitive closure: (1) the Booleans with disjunction as addition, conjunction as multiplication and the closure being *true*; and (2) the natural numbers ($\mathbb{N}$) extended with an element ∞, we let $0_s\ =\ \infty$, $1_s\ =\ 0$, *min* plays the role of $+_s$, addition of natural numbers the role of $\cdot_s$ and the closure is 0.

Matrices To represent the dimensions of matrices we use a type of non-empty binary trees:

```
data Shape : Set where
  L : Shape
  B : (s₁ s₂ : Shape) → Shape
```

This representation follows the structure of the matrix representation more closely than natural numbers and we can easily compute the corresponding natural number:

```
toNat : Shape → ℕ; toNat L = 1; toNat (B l r) = toNat l + toNat r
```

while the other direction is slightly more complicated because we want a somewhat balanced tree and we have no representation for 0.

Matrices are parametrised by the type of elements they contain and indexed by a *Shape* for each dimension. We use a datatype M with four constructors: *One*, *Row*, *Col*, and Q. The first *One* lifts an element into a 1-by-1 matrix:

```
data M (a : Set) : (rows cols : Shape) → Set where
  One :                                a → M a L L
```

Row and column matrices are built from smaller matrices which are either 1-by-1 matrices or further row respectively column matrices

$$\begin{aligned} &Row\ :\ \{c_1\ c_2\ :\ Shape\} \to && M\ a\ L\ c_1 \to M\ a\ L\ c_2 \to M\ a\ L\ (B\ c_1\ c_2) \\ &Col\ :\ \{r_1\ r_2\ :\ Shape\} \to && M\ a\ r_1\ L \to M\ a\ r_2\ L \to M\ a\ (B\ r_1\ r_2)\ L \end{aligned}$$

and matrices of other shapes are built from 2×2 smaller matrices

$$\begin{aligned} Q\ :\ \ \{r_1\ r_2\ c_1\ c_2\ :\ Shape\} \to\ & M\ a\ r_1\ c_1 \to M\ a\ r_1\ c_2 \to \\ & M\ a\ r_2\ c_1 \to M\ a\ r_2\ c_2 \to \\ & M\ a\ (B\ r_1\ r_2)\ (B\ c_1\ c_2) \end{aligned}$$

This matrix representation allows for simple formulations of matrix addition, multiplication, and as we will see also the transitive closure of a matrix.

Transitive closure In [3] Lehmann presents a definition of the closure on square matrices, $A^* = 1 + A \cdot A^*$: Given

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

the transitive closure of A is defined inductively as

$$A^* = \begin{bmatrix} A_{11}^* + A_{11}^* \cdot A_{12} \cdot \Delta^* \cdot A_{21} \cdot A_{11}^* & A_{11}^* \cdot A_{12} \cdot \Delta^* \\ \Delta^* \cdot A_{21} \cdot A_{11}^* & \Delta^* \end{bmatrix}$$

where $\Delta = A_{22} + A_{21} \cdot A_{11}^* \cdot A_{12}$ and the base case is the 1-by-1 matrix where we use the transitive closure of the element of the matrix: $[s]^* = [s^*]$.

We have encoded this definition of closure in Agda and implemented a constructive correctness proof using structural induction and equational reasoning. The full development of around 2500 lines of literate Agda code (including this abstract) is available on GitHub (`https://github.com/DSLsofMath/FLABloM`).

Conclusions We have presented an algebraic structure useful for (block) matrix computations and implemented and proved correctness of transitive closure. Compared to [1] our implementation handles arbitrary matrix dimensions but is restricted to semi-rings. Future work would be to extend the proof to cover both arbitrary dimensions and the more general semi-near-ring structure which would allow parallel parsing as an application.

References

[1] Jean-Philippe Bernardy and Patrik Jansson. Certified context-free parsing: A formalisation of Valiant's algorithm in Agda. *Logical Methods in Computer Science*, 2016. Accepted 2015-12-22 for publication in LMCS. Available from `http://arxiv.org/abs/1601.07724`.

[2] Stephen Dolan. Fun with semirings: A functional pearl on the abuse of linear algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 101–110, New York, NY, USA, 2013. ACM.

[3] Daniel J. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76, 1977.

[4] L.G. Valiant. General context-free recognition in less than cubic time. *J. of computer and system sciences*, 10(2):308–314, 1975.

Answer Set Programming in Intuitionistic Logic

Aleksy Schubert and Paweł Urzyczyn[1]

University of Warsaw
[alx,urzy]@mimuw.edu.pl

Abstract

We propose the first interpretation of propositional answer set programming (ASP) in terms of intuitionistic proof theory, in particular in terms of simply typed lambda calculus. While connections between ASP and intuitionistic logic are well-known, they usually take the form of characterizations of stable models with the help of some intuitionistic theories represented by specific classes of Kripke models. As such the known results are model-theoretic rather than proof-theoretic. In contrast, we propose an interpretation of ASP in terms of constructive proofs.

Answer Set Programming (ASP) is a programming paradigm originated from logic programming with negation understood as "fixpoint" [1, 2, 3]. The paradigm proved useful by reducing search problems to so caled stable models of declarative programs.

It has been observed long ago [5, 6, 7] that ASP can be interpreted using certain intuitionistic theories or intermediate logics, of which *equilibrium logic* of Pearce [7] is the most fundamental example. This is commonly summarized as applying intuitionistic logic to ASP. But what is actually done is representing answer sets as specific Kripke models (often two-state models).

We propose another way to interpret ASP in intuitionistic logic, namely we want to represent ASP inference (entailment in stable models) by intuitionistic provability. We use the ordinary intuitionistic logic (without any additional axioms) for this purpose. Our proof-theoretical account brings new insights to the operational semantics of answer set programming. This seems to suggest that intuitionistic provers can serve as natural ASP-solvers. Yet differently, perhaps intuitionistic logic (together with its highly intuitive and natural lambda-notation) should itself be seen as a programming paradigm, competitive to ASP?

In the present note we only consider propositional ASP and we reduce it to the implicational fragment (in particular no negation is needed) of propositional intuitionistic logic (IPC). We believe this approach should turn out very flexible by easily accomodating various extensions of ASP. In particular, a similar approach should be applicable to first-order ASP.

Propositional ASP: A *literal* is a propositional atom or its negation. A *clause* is an expression of the form $X :- X_1, \ldots, X_n$, where X is a propositional atom and $X_1, \ldots, X_n$ are literals. A *program* is a finite set of clauses. A *model* is a set M of atoms, identified with a valuation v_M such that $v_M(X) = \mathit{true}$ if and only if $X \in M$. Given a program P and a model M, we obtain P^M from P as follows. First, for all $X \notin M$, we delete $\neg X$ from the rhs of all clauses of P. Then, for all $X \in M$, delete all clauses of P with $\neg X$ at the rhs. Now the *interpretation* of P under M, denoted $I(P, M)$, is defined as the least fixed point of the operator:

$$F(\mathcal{S}) = \mathcal{S} \cup \{X \mid \text{there is a clause } X :- X_1, \ldots, X_n \text{ in } P^M \text{ such that all } X_i \text{ are in } \mathcal{S}\}.$$

A model M is a *stable* model of P (or it is an *answer set* for P) iff $M = I(P, M)$. We also say that P *entails* an atom X *under SMS*, written $P \models_{SMS} X$, iff every stable model of P satisfies X. It is known [2, 4] that the existence of a stable model and the entailment under SMS are, respectively, NP and *co*-NP complete problems.

Intuitionistic logic: We consider formulas of minimal propositional logic with $\to$ as the only propositional connective. That is, our formulas are just simple types, and intuitionistic proofs can be identified with simply-typed lambda-terms.

Given a program P we define an implicational formula φ such that P entails X under stable model semantics if and only if φ is intuitionistically provable. Let $X_1, \ldots, X_n$ be all propositional atoms occurring in P, including X. Without loss of generality we assume that X_i and $\neg X_i$ never occur together in the rhs of a clause.

Assume that P consists of m clauses, numbered from 1 to m. The vocabulary of φ consists of atoms $0, 1, \ldots, n, X_1, \ldots, X_n, \overline{X}_1, \ldots, \overline{X}_n, X_1!, \ldots, X_n!, X_1?, \ldots, X_n?, A, B, K_1, \ldots, K_m$.

The formula φ has the form $\psi_1 \to \cdots \to \psi_d \to 0$. The assumption formulas $\psi_1, \ldots, \psi_d$ are defined below. The first n assumptions are as follows:

$$\psi_1 = (X_1 \to 1) \to (\overline{X}_1 \to 1) \to 0, \qquad \ldots \qquad \psi_n = (X_n \to n) \to (\overline{X}_n \to n) \to n - 1.$$

The next m assumptions are the clauses of P, where every atom X is replaced by $X!$ and every $\neg X$ is replaced by $\overline{X}$. Then we have three formulas: $X \to n$, $A \to n$, $B \to n$, with target n. For every $i = 1, \ldots, n$, there are assumptions $\overline{X}_i \to X_i! \to A$ and $X_i \to X_i? \to B$. For $i = 1, \ldots, n$, there is an assumption $(X_i? \to K_{s_1}) \to \cdots \to (X_i? \to K_{s_{r_i}}) \to X_i?$, where $s_1, \ldots, s_{r_i}$ are (numbers of) all the clauses of P with target X_i.

If the atom X_i occurs at the rhs of the sth clause of P then there is an assumption $X_i? \to K_s$. And if $\neg X_i$ occurs at the rhs of the sth clause of P then there is an assumption $X_i \to K_s$.

The formulas naturally reflect the semantical character of the definition of ASP. Their intended meaning is as follows: Formulas $\psi_1, \ldots, \psi_n$ choose a binary valuation of atoms in a certain model. In order to prove 0 one proves n under assumptions representing every model. Now there are three ways in which the entailment $P \models X$ holds in a stable model: either X holds, or the model is unstable because P is unsound (proves too much) or the model is unstable because P is incomplete (does not prove what is needed). The assumption $X \to n$ can be used to complete the proof when X holds in the model. The two other possibilities are represented by A and B, respectively. One proves A when P is unsound for our model: it forces some X_i to hold (that is, $X_i \in I(P, M)$), but $\overline{X}_i$ is chosen. Proving B means that P^M is unable to derive an atom X_i which is present in the model. The propositional atoms $X?$ and K_s are understood as "X has no proof" and "the target of the sth clause has no proof", respectively. The formula with target $X_i?$ should be understood as follows: no clause with target X_i has a proof unless such a proof recursively refers to the proof goal X_i.

References

[1] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, December 2011.

[2] Phokion G. Kolaitis and Christos H. Papadimitriou. Why not negation by fixpoint? In *PODS '88*, pages 231–239, ACM, 1988.

[3] Vladimir Lifschitz. What is answer set programming? In *AAAI'08*, pages 1594–1597. AAAI Press, 2008.

[4] Wiktor Marek and Mirosław Truszczyński. Autoepistemic logic. *J. ACM*, 38(3):587–618, July 1991.

[5] Mauricio Osorio, Juan A. Navarro, and José Arrazola. Applications of intuitionistic logic in answer set programming. *Theory Pract. Log. Program.*, 4(3):325–354, May 2004.

[6] David Pearce. Stable inference as intuitionistic validity. *The Journal of Logic Programming*, 38(1):79–91, 1999.

[7] David Pearce. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):3–41, 2006.

The Use of the Coinduction Hypothesis in Coinductive Proofs

Anton Setzer

Dept. of Computer Science, Swansea University, Swansea, UK
a.g.setzer@swan.ac.uk

Abstract

For inductively defined sets proofs by induction are carried out using the induction hypothesis. We demonstrate how such a methodology can be applied as well to proofs by coinduction, which can make use of the coinduction hypothesis. We show as well how to apply this methodology to proofs of bisimilarity on transition systems.

The goal of this talk is to transfer the way of carrying out coinductive proofs in type theoretic theorem provers to reasoning in ordinary mathematics. We will use standard set theoretic notion, however use $n : \mathbb{N}$ instead of $n \in \mathbb{N}$.

When reasoning about inductively defined sets we are used to argue informally while referring to the induction hypothesis. When for instance showing $\forall x, y, z : \mathbb{N}.(x + y) + z = x + (y + z)$, we do not first define a set $R := \{z : \mathbb{N} \mid \forall x, y.(x + y) + z = x + (y + z)\}$ and then argue that R is closed under 0 and successor. Instead we show $(x + y) + 0 = x + (y + 0)$ and prove $(x + y) + \mathrm{S}(z) = x + (y + \mathrm{S}(z))$ by using the induction hypothesis $(x + y) + z = x + (y + z)$. Both forms of reasoning are equivalent, but the latter is more light weight and easier to use.

When proving properties about coinductively defined sets, i.e. final coalgebras, we are currently usually following principles which are similar to using for natural numbers the set R being closed under $0, \mathrm{S}$ as above. For instance when showing that two elements of a labelled transition system are bisimilar, one defines a relation on pairs of states of the transition system and shows that it is a bisimulation relation. This makes coinductive reasoning difficult to use.

In [2] we have introduced very general schemata for reasoning corecursively and coinductively by using a coinduction hypothesis in a similar way as one reasons primitive recursively and inductively. The theory was developed for the coinductive version of the Petersson Synek Trees, which capture using containerisation a large class of strictly positive indexed coinductively defined sets. In this talk we will present some examples of how to apply this to more simple examples. The use of the coinduction hypothesis is simplified by making use of the definition of coinductively defined sets as given by their elimination rules [1].

Consider the example of the set of increasing streams $\mathrm{IncStream}(n)$, which is the coinductively defined set indexed over $\mathbb{N}$ given by the eliminators (observations)

$$\begin{array}{lcl} \mathrm{head} & : & (n : \mathbb{N}) \to \mathrm{IncStream}(n) \to \mathbb{N}_{\geq n} \\ \mathrm{tail} & : & (n : \mathbb{N}) \to (s : \mathrm{IncStream}(n)) \to \mathrm{IncStream}(\mathrm{head}(n, s) + 1) \end{array}$$

where $\mathbb{N}_{\geq n} := \{m : \mathbb{N} \mid m \geq n\}$.

The schema for *corecursion* expresses that we can define, assuming a set X and $i : X \to \mathbb{N}$ a function $f : (x : X) \to \mathrm{IncStream}(i(x))$, provided we define for $x : X$

$$\begin{array}{lcccl} \mathrm{head}(i(x), f(x)) & = & m & : & \mathbb{N}_{\geq i(x)} \\ \mathrm{tail}(i(x), f(x)) & = & s & : & \mathrm{IncStream}(m + 1) \end{array}$$

where m and s depend on x, and s can be defined either as an element of $\mathrm{IncStream}(m + 1)$ known before defining f or s can be given by the *corecursion hypothesis* $f(x')$ for some x' s.t. $i(x') = m + 1$.

For instance we can define

$$\begin{array}{lcl}\mathrm{inc}, \mathrm{inc}', \mathrm{inc}'' & : & (n : \mathbb{N}) \to \mathrm{IncStream}(n)\\ \mathrm{head}(n, \mathrm{inc}(n)) & = & \mathrm{head}(n, \mathrm{inc}'(n)) = \mathrm{head}(n, \mathrm{inc}''(n)) = n\\ \mathrm{tail}(n, \mathrm{inc}(n)) & = & \mathrm{inc}(n+1)\\ \mathrm{tail}(n, \mathrm{inc}'(n)) & = & \mathrm{inc}''(n+1)\\ \mathrm{tail}(n, \mathrm{inc}''(n)) & = & \mathrm{inc}'(n+1)\end{array}$$

This definition can be made to confirm with the schema by replacing the simultaneous definition of $\mathrm{inc}, \mathrm{inc}', \mathrm{inc}''$ by one function combined : $((n : \mathbb{N}) \times \{\mathrm{inc}, \mathrm{inc}', \mathrm{inc}''\}) \to \mathrm{IncStream}(n)$.
When we have final coalgebras, then we get as well the schema of coinduction. In case of IncStream it is as follows: Assume $i : X \to \mathbb{N}$ and $f, g : (x : X) \to \mathrm{IncStream}(i(x))$. We can prove $\forall x : X.f(x) = g(x)$ by showing the following:

$$\begin{array}{lcl}\forall x : X. \quad \mathrm{head}(i(x), f(x)) & = & \mathrm{head}(i(x), g(x))\\ \forall x : X. \quad \mathrm{tail}(i(x), f(x)) & = & \mathrm{tail}(i(x), g(x))\end{array}$$

where for proving $\mathrm{tail}(i(x), f(x)) = \mathrm{tail}(i(x), g(x))$ we can use the fact that $\mathrm{tail}(i(x), f(x)) = f(x')$ and $\mathrm{tail}(i(x), g(x)) = g(x')$ for some x' s.t. $i(x') = \mathrm{head}(i(x), f(x)) + 1$, and by using the *coinduction hypothesis* $f(x') = g(x')$.
For instance we can prove by coinduction on IncStream

$$(\forall n : \mathbb{N}.\mathrm{inc}(n) = \mathrm{inc}'(n)) \wedge (\forall n : \mathbb{N}.\mathrm{inc}(n) = \mathrm{inc}''(n))$$

as follows:

$$\begin{array}{lcllcllcl}\mathrm{head}(\mathrm{inc}(n)) & = & n & = & \mathrm{head}(\mathrm{inc}'(n)) & = & n & = & \mathrm{head}(\mathrm{inc}''(n))\\ \mathrm{tail}(\mathrm{inc}(n)) & = & \mathrm{inc}(n+1) & \overset{\text{co-IH}}{=} & \mathrm{inc}''(n+1) & = & \mathrm{tail}(\mathrm{inc}'(n)) & & \\ \mathrm{tail}(\mathrm{inc}(n)) & = & \mathrm{inc}(n+1) & \overset{\text{co-IH}}{=} & \mathrm{inc}'(n+1) & = & \mathrm{tail}(\mathrm{inc}''(n)) & & \end{array}$$

The above principle applies as well to proofs of bisimilarity $\sim$ for labelled transition systems. Restricting ourselves to the unlabelled case, let a transition system be given by a set T and a relation $\longrightarrow \subseteq \mathrm{T} \times \mathrm{T}$. The schema for proving bisimilarity on $(\mathrm{T}, \longrightarrow)$ is as follows: Assume $f, g : X \to \mathrm{T}$. We can prove $\forall x \in X.f(x) \sim g(x)$ by giving:

- for $f(x) \longrightarrow t$ a t' s.t. $g(x) \longrightarrow t'$ and $t \sim t'$,
- for $g(x) \longrightarrow t'$ a t s.t. $f(x) \longrightarrow t$ and $t \sim t'$.

where in case $t = f(x')$ and $t' = g(x')$ for some x' we are allowed to use for proving $t \sim t'$ the *coinduction-hypothesis* $f(x') \sim g(x')$.
For instance, let $\mathrm{T} := \{*\} \cup \mathbb{N}$, $\longrightarrow$ given by $* \longrightarrow *$, $n \longrightarrow n + 1$.
We show $\forall n \in \mathbb{N}.* \sim n$:

- Assume $* \longrightarrow x$. Then $x = *$. $n \longrightarrow n + 1$ and by co-IH $* \sim n + 1$.
- Assume $n \longrightarrow x$. Then $x = n + 1$. $* \longrightarrow *$ and by co-IH $* \sim n + 1$.

References

[1] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In R. Giacobazzi and R. Cousot, editors, *Proceedings of POPL '13*, pages 27–38. ACM, 2013.

[2] A. Setzer. How to reason coinductively informally. To appear in: Reinhard Kahle, Thomas Strahm, and Thomas Studer (Eds.): Advances in Proof Theory, Birkhäuser, 2016.

Automorphisms of Types, Cayley Graphs and Representation of Finite Groups *

Sergei Soloviev

IRIT, University of Toulouse-3,
118, route de Narbonne, 31062, Toulouse, France,
soloviev@irit.fr

Isomorphisms of types are represented by λ-terms $t : A \to B$ such that there exists $t^{-1} : B \to A$ and $t^{-1} \circ t \equiv id_A, t \circ t^{-1} \equiv id_B$. (For detailed definitions and history see, *e.g.*, [2].)

An isomorphism is called an automorphism if $A = B$. For example, for $A = a \to (a \to a)$ there are two automorphisms: the identity, and the permutation of premises. Obviously, the automorphisms $t : A \to A$ form a group w.r.t. composition.

The automorphisms and automorphism groups of types were not studied in [2], but the methods developed there permit to obtain a straightforward proof of the following theorem:

Theorem 1 A finite group G is isomorphic to the group of automorphisms of some type A of simply typed λ-calculus with surjective pairing and terminal object iff it is isomorphic to the group of automorphisms of a finite rooted tree.

It turns out that using either the types of system F (see, e.g., [2]) or dependent product kinds (of, say, logical framework LF, see [4] or [6]) it is possible to represent *any* finite group. Let G be a finite group with the set of generators $S \subseteq G$. Let $C_S(G)$ denote its Cayley colored graph [8].

Proposition. (See [8], theorem 4-8.) The subgroup of color-preserving automorphisms of $Aut(C_S(G))$ is isomorphic to G.

Theorem 2. It is possible to construct a type $T_S(G)$ of system F (a kind $K_S(G)$ of LF) such that:

(i)The group of automorphisms of the type $T_S(G)$ is isomorphic to the group of color-preserving automorphisms of $C_S(G)$ and thus isomorphic to G.

(ii) The group of automorphisms of the type $K_S(G)$ is isomorphic to the group of color-preserving automorphisms of $C_S(G)$ and thus isomorphic to G.

The proof uses an explicit construction of $T_S(G)$ and $K_S(G)$. An important difference with the simply typed case is exploited: the fact that the equality of types in system F (or LF) is non-trivial, for example, it includes the α-conversion.

Example. The following types of system F are α-equal:

$$\forall X.\forall Y.\forall Z.((X \to Y) \to (Y \to Z) \to (Z \to X) \to \forall U.U)$$

$$\forall Z.\forall X.\forall Y.((Z \to X) \to (X \to Y) \to (Y \to Z) \to \forall U.U)$$

$$\forall Y.\forall Z.\forall X.((Y \to Z) \to (Z \to X) \to (X \to Y) \to \forall U.U).$$

The non-trivial automorphisms are obtained by *simultaneous* application of the same permutation to the quantifiers and the premises of implication. The group of automorphisms is the cyclic group C_3. (In general the construction is more complex and uses an encoding of colours of the Cayley graph.)

Applications. The types isomorphic to a given type and their isomorphisms form a groupoid (in cases of simple types, types of system F and kinds of LF it is finite). The

*This work was partially supported by the Climt project, ANR-11-BS02-016-02.

study of automorphism groups of types is part of the study of this groupoid, and may be of interest for foundational research, for example, the Homotopy Type Theory [1]. Another, more practical application is suggested by the observation that since automorphisms do not change data types, but change their elements, they may be of use in cryptography.

Future research. It would be of interest to study automorphisms in other type systems, for example the simply typed calculi with empty and sum types [3]. Consideration of retractions (cf. [7]) may be interesting if one wants to go beyond groups and groupoids as algebraic structures to be represented.

Acknowledgements. A paper that contains the proofs of these results is actually submitted to MSCS [5]. I would like to thank Ralph Matthes for useful remarks concerning the early version of this paper, and anonymous referees for remarks concerning this abstract.

References

[1] *Homotopy Type Theory: Univalent Foundations of Mathematics*. IAS, Princeton, 2013.

[2] R. Di Cosmo. *Isomorphisms of types: from lambda-calculus to information retrieval and language design*. Birkhauser, 1995.

[3] M. Fiore. Isomorphisms of generic recursive polynomial types. In *POPL '04*, pages 77–88, New York, USA, 2004. ACM.

[4] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.

[5] S. Soloviev. Automorphisms of types in certain type theories and representation of finite groups. *Mathematical Structures in Computer Science*, submitted, December 2015.

[6] S. Soloviev. On isomorphism of dependent products in a typed logical framework. In *TYPES 2014*, LIPICS, pages 275–288, Schloss Dagstuhl, 2015. Leibniz-Zentrum für Informatik.

[7] C. Stirling. Proof systems for retracts in simply typed lambda calculus. In *Proceedings of ICALP*, volume 2, pages 398–409, 2013.

[8] A. T. White. *Graphs, Groups and Surfaces*. North-Holland, 1984.

My Experience with the Lean Theorem Prover

Jakob von Raumer

University of Nottingham, Nottingham, United Kingdom
psxjv4@nottingham.ac.uk

Lean is a dependently typed theorem prover[1] which is developed mainly by Leonardo de Moura at Microsoft Research. Even though the project is very young – the development started in 2013 – it already provides a very usable language for formalizations of all kinds. It allows for proof irrelevant reasoning as well as homotopy type theory and it has an extensive library for both of these modes which is mainly written by Jeremy Avigad and others at Carnegie Mellon University, Pittsburgh.

I will first give a little tour through the features of Lean, presenting its syntax and current means of automation, and comparing it to other widespread provers like Coq, Agda and Isabelle. The features presented include:

- Defining inductive types,
- defining and extending structures (record types),
- writing proof terms in direct syntax or using tactics, and
- using type classes.

Then, I will show in some examples how these features were used to build Lean's (proof irrelevant) standard library and its homotopy type theory library.

Afterwards, I will describe my experience formalizing double gropuoids with thin structure and crossed modules as algebraic structures to contain the first and second homotopy groupoids of a higher type. I will focus on the difficulties to reconcile abstraction height and proof checker performance, which can be an issue in any larger formalization project.

[1] http://leanprover.github.io

Author Index