Types for Proofs and Programs

19th TYPES Meeting

Toulouse, France

April 22–26, 2013

TYPES 2013

Book of Abstracts

Collected by Ralph Matthes (IRIT, CNRS)

Preface

This is the collection of the abstracts of the *19th Conference "Types for Proofs and Programs", TYPES 2013*, to take place in Toulouse, France, 22–26 April 2013.

The Types Meeting is 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. Since 1992, Types Meetings have been annual workshops of several multilateral scientific EU-financed projects, of which the Types Project was the most recent. Although organized in Europe, the meetings were always open internationally, as evidenced by invited speakers and authors of contributed talks.

TYPES 2013 was intended to be a conference in our traditional workshop style, and the selection of contributions was based on abstracts of up to two pages. Abstracts were generally reviewed by three members of the program committee (in some cases helped by external reviewers whose names should not be disclosed here):

- José Espírito Santo, University of Minho, Braga, Portugal
- Herman Geuvers, Radboud University Nijmegen, Netherlands
- Silvia Ghilezan, University of Novi Sad, Serbia
- Hugo Herbelin, PPS, INRIA Rocquencourt-Paris, France
- Martin Hofmann, Ludwig-Maximilians-Universität München, Germany
- Zhaohui Luo, Royal Holloway, University of London, UK
- Ralph Matthes, IRIT, CNRS and Univ. de Toulouse, France (co-chair)
- Marino Miculan, University of Udine, Italy
- Bengt Nordström, Chalmers University of Technology, Göteborg, Sweden
- Erik Palmgren, Stockholm University, Sweden
- Andy Pitts, University of Cambridge, UK
- Sergei Soloviev, IRIT, Univ. de Toulouse, France (co-chair)
- Paweł Urzyczyn, University of Warsaw, Poland
- Tarmo Uustalu, Institute of Cybernetics, Tallinn Technical University, Estonia

The present volume provides final versions of the abstracts of three invited speakers (chosen by the program committee)

- Steve Awodey, School of Mathematics, Institute for Advanced Study, Princeton, U.S.A. & Department of Philosophy, Carnegie Mellon University, Pittsburgh, U.S.A.
- Lars Birkedal, Department of Computer Science, Aarhus University, Denmark
- Ulrich Kohlenbach, Department of Mathematics, Technische Universität Darmstadt, Germany

and 34 contributed talks. We have to regret that one accepted talk had to be withdrawn by its author since he could not come to the conference.

Acknowledgements

Thanks go to all the authors of abstract submissions, whether accepted or not. They were the raw material to shape this scientific meeting. A big thank you to the invited speakers for accepting this invitation to come to Toulouse. And, of course, the effort of the program committee members is gratefully acknowledged and also the anonymous external reviewers.

I am much obliged to Andrej Voronkov and the support team of EasyChair for this marvellous tool free of charge that is very beneficial for the program committee and in particular for (co-)chairing it and also still quite useful for preparing this volume.

A note on the PDF version of this document: all the given links to web pages were verified yesterday.

Finally, and very importantly, the following institutions generously helped with funding and/or in providing lecture halls and services that TYPES 2013 could take place (in alphabetic order):

- Institut de Recherche en Informatique de Toulouse (IRIT), `http://www.irit.fr/`
- Région Midi-Pyrénées, `http://www.midipyrenees.fr/`
- Structure Fédérative de Recherche en Mathématiques et en Informatique de Toulouse (FREMIT), `http://www.irit.fr/FREMIT/`
- Université Paul Sabatier, Toulouse III, `http://www.univ-tlse3.fr/`
- Université Toulouse 1 Capitole, `http://www.ut-capitole.fr/`

April 12, 2013
Ralph Matthes, IRIT (CNRS and Université de Toulouse, France)

Invited Talks

Higher Inductive Types in Homotopy Type Theory

Steve Awodey[1]*

Institute for Advanced Study
School of Mathematics
Princeton, NJ 08540 USA

Carnegie Mellon University
Department of Philosophy
Pittsburgh, PA 15213 USA

Homotopy Type Theory (HoTT) refers to the homotopical interpretation [1] of Martin-Löf's intensional, constructive type theory (MLTT) [5], together with several new principles motivated by that interpretation. Voevodsky's *Univalent Foundations* program [6] is a conception for a new foundation for mathematics, based on HoTT and implemented in a proof assistant like Coq [2].

Among the new principles to be added to MLTT are the Univalence Axiom [4], and the so-called *higher inductive types* (HITs), a new idea due to Lumsdaine and Shulman which allows for the introduction of some basic spaces and constructions from homotopy theory. For example, the n-dimensional spheres S^n can be implemented as HITs, in a way analogous to the implementation of the natural numbers as a conventional inductive type. Other examples include the unit interval; truncations, such as bracket-types $[A]$; and quotients by equivalent relations or groupoids. The combination of univalence and HITs is turning out to be a very powerful and workable system for the formalization of homotopy theory, with the recently given, formally verified proofs of some fundamental results, such as determinations of various of the homotopy groups of spheres by Brunerie and Licata. See [3] for much work in progress

After briefly reviewing the foregoing developments, I will give an impredicative encoding of certain HITs on the basis of a new representation theorem, which states that every type of a particular kind is equivalent to its double dual in the space of coherent natural transformations. A realizability model is also provided, establishing the consistency of impredicative HoTT and its extension by HITs.

References

[1] S. Awodey and M. A. Warren, *Homotopy theoretic models of identity types*, Mathematical Proceedings of the Cambridge Philosophical Society 146 (2009), no. 1, 45-55.

[2] The Coq proof assistant, `http://coq.inria.fr`

[3] The IAS Univalent Foundations wiki, `http://uf-ias-2012.wikispaces.com`.

[4] C. Kapulkin, P. Lumsdaine, V. Voevodsky: *The simplicial model of univalent foundations*, on the arXiv as arXiv:math/1211.2851, 2012.

[5] P. Martin-Löf, *An intuitionistic theory of types*, Twenty-five years of constructive type theory (Venice, 1995), Oxford Univ. Press, New York, 1998, pp. 127-172.

[6] V. Voevodsky, *Univalent foundations project*, Modified version of an NSF grant application, `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations.html`, 2010.

*NSF Grant DMS-1001191 and Air Force OSR Grant 11NL035

Charge! a framework for higher-order separation logic in Coq.

Lars Birkedal

Department of Computer Science
Aarhus University
Aarhus, Denmark

Higher-order separation logic (HOSL) is an extension of separation logic that allows for quantification over predicates in both the assertion logic (the logic of pre- and post-conditions) and the specification logic (the logic of Hoare triples). Higher-order separation logic has proved useful for modular reasoning about programs that use shared mutable data structures and data abstraction, via quantification over resource invariants, and for reasoning about various forms of higher-order programming (higher-order functions, code pointers, interfaces in object-oriented programming).

In this talk I will describe our work on Charge!, a separation-logic verification tool implemented in Coq, that aims to

1. prove full functional correctness of Java-like programs using higher-order separation logic,
2. produce machine-checkable correctness proofs,
3. work as close as possible to how informal separation logic proofs are carried out on pen and paper, and
4. automate tedious first-order reasoning where possible.

Joint work with Jesper Bengtson, Jonas B. Jensen, Hannes Mehnert, Peter Sestoft, and Filip Sieczkowski.

Types in Proof Mining

Ulrich Kohlenbach

Technische Universität Darmstadt, Fachbereich Mathematik
Schlossgartenstraße 7, 64289 Darmstadt, Germany
kohlenbach@mathematik.tu-darmstadt.de

Abstract

During the last 20 years a new applied form of proof theory (sometimes referred to as 'proof mining') has been developed which uses proof-theoretic transformations to extract hidden quantitative and computational information from given (prima facie ineffective) proofs ([6]). The historical roots of this development go back to the 50's and the pioneering work on 'unwinding of proof' done by G. Kreisel ([10]). The modern 'proof mining' paradigm has been most systematically pursued in the context of nonlinear analysis, ergodic theory and fixed point theory in recent years. The main proof-theoretic techniques used are extensions and novel forms of functional interpretation that are based on Gödel's famous 1958 'Dialectica'-interpretation ([1, 7]). All these interpretations are formulated in languages of functionals of finite types.

Until 2000, applications of proof mining in analysis mainly concerned the context of Polish spaces and continuous functions between them which can be represented by the Baire space $\mathbb{N}^{\mathbb{N}}$ and functionals $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$ and so finite types over the base type $\mathbb{N}$ for natural numbers were sufficient.

Starting with 2001 ([4]) a number of applications in metric fixed point theory emerged which deal with theorems that hold in general classes of spaces such as all Banach spaces or all Hilbert spaces etc. In these applications, effective bounds of an amazing uniformity were extracted in the sense that the bounds did not depend on metrically bounded parameters which in the context of Polish space usually can only be expected under strong compactness assumptions. In fact, it is the very feature of the proofs in question not to use any separability assumptions on the spaces in question which makes this possible.

Starting in 2005, general so-called logical metatheorems have been developed which explain these applications and paved the way for numerous new applications ([5, 3, 6, 2]). In order to faithfully reflect the absence of any use of separability, this requires a formal framework different from the usual ones used in proof theory, constructive mathematics, reverse mathematics or computable analysis which usually represent (complete) metric or Banach spaces as the completion of a countable dense subset. In our framework, we instead add abstract metric structures X (or $X_1, \dots, X_n$) as new base types to the language together with the appropriate constants (such as a (pseudo-)metric d_X) and the appropriate axioms and consider all finite types over $\mathbb{N}, X_1, \dots, X_n$. Equality for such a new base type X with pseudo-metric d_X is a defined notion $x =_X y := d_X(x, y) =_{\mathbb{R}} 0$ reflecting that we work over the metric space induced by the pseudo-metric d_X. Here real numbers are represented via the usual Cauchy representation and $=_{\mathbb{R}} \in \Pi^0_1$ is the corresponding equivalence relation on the space $\mathbb{N}^{\mathbb{N}}$ of Cauchy names. It is the interplay between the world of abstract structures X and the world of represented concrete Polish spaces such as $\mathbb{R}$ which requires considerable care.

As well-known from functional interpretation, the ability of the latter to unwind proofs based on classical logic (by satisfying the so-called Markov principle) make it necessary to put a severe restriction on the use of the axiom of extensionality (i.e. the axiom stating that functionals

respect the extensionally defined equality) which has to be replaced by a weak extentsionality rule. Whereas this is not a real restriction in the context of Polish spaces (due to the well-known elimination-of-extensionality method of Gandy and Luckhardt), this is rather different in our context where already the extensionality of objects f of type $X \to X$ is too strong to be included as an axiom. While in many cases, this extensionality follows from assumptions on f (such as being Lipschitzian), one has to rely on weak extensionality in other cases.

Although the theorems from mathematics to be studied always only use types of very low degree, the proof-transformations based on functional interpretations make use of the whole hierarchy of finite types. Moreover, it is via these types that the uniform bounding data to be extracted from the proof are being controlled throughout the proof via novel forms of majorization. Let ρ be a finite type over $\mathbb{N}, X_1, \ldots, X_n$ and $\widehat{\rho}$ be the result of replacing all occurrences of $X_1, \ldots, X_n$ in ρ by $\mathbb{N}$. Then the objects serving as majorants for objects of type ρ are of type $\widehat{\rho}$ and all the computations take place on these majorants, i.e. on objects which have a finite type over $\mathbb{N}$ so that usual computability theory applies irrespectively of whether the structures $X_1, \ldots, X_n$ carry any computability notion.

We will give a survey on these developments and – as time permits – present some recent applications to nonlinear ergodic theory dealing with explicit rates of asymptotic regularity and metastability as well as algorithmic learning information ([8, 9, 11]).

References

[1] Jeremy Avigad and Solomon Feferman. Gödel's functional ('Dialectica') interpretation. In Samuel R. Buss, editor, *Handbook of Proof Theory*, pages 337–405. Elsevier, 1998.

[2] Eyvind Martol Briseid. Logical aspects of rates of convergence in metric spaces. *J. Symb. Log.*, 74:1401–1428, 2009.

[3] Philipp Gerhardy and Ulrich Kohlenbach. General logical metatheorems for functional analysis. *Trans. Amer. Math. Soc.*, 360:2615–2660, 2008.

[4] Ulrich Kohlenbach. A quantitative version of a theorem due to Borwein-Reich-Shafrir. *Numer.Funct.Anal.Optimiz.*, 22:641–656, 2001.

[5] Ulrich Kohlenbach. Some logical metatheorems with applications in functional analysis. *Trans. Amer. Math. Soc.*, 357:89–128, 2005.

[6] Ulrich Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics.* Springer Monographs in Mathematics. Springer Verlag, 2008.

[7] Ulrich Kohlenbach. Gödel's functional interpretation and its use in current mathematics. In M. et al. Baaz, editor, *Kurt Gödel and the Foundations of Mathematics. Horizons of Truth*, pages 223–267. Cambridge University Press, New York, 2011.

[8] Ulrich Kohlenbach and Laurenţiu Leuştean. Effective metastability for Halpern iterates in CAT(0) space. *Advances in Mathematics*, 231:2526–2556, 2012.

[9] Ulrich Kohlenbach and Laurenţiu Leuştean. On the computational content of convergence proofs via Banach limits. *Philosophical Transactions of the Royal Society A*, 370:3449–3463, 2012.

[10] Angus Macinytre. The mathematical significance of proof theory. *Philosophical Transactions of the Royal Society A*, 363:2419–2435, 2005.

[11] Pavol Safarik. A quantitative nonlinear strong ergodic theorem for Hilbert spaces. *J. Math. Analysis Appl.*, 391:26–37, 2012.

Contributed Talks

Copatterns: Programming Infinite Structures by Observations

Andreas Abel

Institut für Informatik, Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, DEUTSCHLAND

Inductive datatypes provide mechanisms to define finite data such as finite lists and trees via constructors and allow programmers to analyze and manipulate finite data via pattern matching. In this talk, we present a dual approach for working with infinite data structures such as streams. Infinite data inhabits coinductive datatypes which denote greatest fixpoints. Unlike finite data which is defined by constructors we define infinite data by observations. Dual to pattern matching, a tool for analyzing finite data, we develop the concept of copattern matching, which allows us to synthesize infinite data. This leads to a symmetric language design where pattern matching on finite and infinite data can be mixed.

We present a core language for programming with infinite structures by observations together with its operational semantics based on (co)pattern matching and describe coverage of copatterns. Our language naturally supports both call-by-name and call-by-value interpretations and can be seamlessly integrated into existing languages like Haskell and ML. We prove type soundness for our language and sketch how copatterns open new directions for solving problems in the interaction of coinductive and dependent types.

This is joint work with Brigitte Pientka, David Thibodeau, and Anton Setzer [1].

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23 - 25, 2013*, pages 27–38. ACM Press, 2013.

Update Monads: Cointerpreting Directed Containers

Danel Ahman[1] and Tarmo Uustalu[2]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh,
10 Crichton Street, Edinburgh EH8 9LE, United Kingdom; d.ahman@ed.ac.uk
[2] Institute of Cybernetics, Tallinn University of Technology,
Akadeemia tee 21, 12618 Tallinn, Estonia; tarmo@cs.ioc.ee

Containers are a neat representation of a wide class of set functors. We have previously [1] introduced directed containers as a concise representation of comonad structures on such functors. Here we examine interpreting the *opposite* categories of containers and directed containers. We arrive at a new view of a different (considerably narrower) class of set functors and monads on them, which we call update monads.[1]

A *container* is given by a set S (of shapes) and an S-indexed family of sets P (of positions). Containers form a category **Cont** with a (composition) monoidal structure. Containers interpet into set functors by

$$[\![S, P]\!]^{\mathrm{c}}\, X = \Sigma s : S.\, P\, s \to X$$

The functor $[\![-]\!]^{\mathrm{c}} : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$ is monoidal and fully faithful.

A *directed container* is a container (S, P) together with operations

$\downarrow : \Pi s : S. P\, s \to S$ (subshapes)
$\mathsf{o} : \Pi\{s : S\}.\, P\, s$ (the root)
$\oplus : \Pi\{s : S\}.\, \Pi p : P\, s.\, P\, (s \downarrow p) \to P\, s$ (subshape positions as positions in the global shape)

satisfying the laws

$$\begin{aligned}
&\forall\{s\}.\, s \downarrow \mathsf{o} = s\\
&\forall\{s, p, p'\}.\, s \downarrow (p \oplus p') = (s \downarrow p) \downarrow p'\\
&\forall\{s, p\}.\, p \oplus \{s\}\, \mathsf{o} = p\\
&\forall\{s, p\}.\, \mathsf{o}\, \{s\} \oplus p = p\\
&\forall\{s, p, p', p''\}.\, (p \oplus \{s\}\, p') \oplus p'' = p \oplus (p' \oplus p'')
\end{aligned}$$

so $(P, \mathsf{o}, \oplus)$ is a bit like a monoid (but dependently typed) and $(S, \downarrow)$ like its action on a set. Directed containers are the same as comonoids in the category of containers: **DCont** $\cong$ **Comonoids**(**Cont**). The interpretation of containers into set functors extends into an interpretation of directed containers into comonads via

$$\begin{aligned}
&\varepsilon : \forall\{X\}.\, (\Sigma s : S.\, P\, s \to X) \to X\\
&\varepsilon\, (s, v) = v\, (\mathsf{o}\, \{s\})\\
&\delta : \forall\{X\}.\, (\Sigma s : S.\, P\, s \to X) \to \Sigma s : S.\ P\, s \to \Sigma s' : S.\ P\, s' \to X\\
&\delta\, (s, v) = (s, \lambda p.\, (s \downarrow p, \lambda p'.\, v\, (p \oplus\ p')))
\end{aligned}$$

The functor $[\![-]\!]^{\mathrm{dc}} : \mathbf{DCont} \to \mathbf{Comonads}(\mathbf{Set})$ is the pullback of the functor $[\![-]\!]^{\mathrm{c}} : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$ along $U : \mathbf{Comonads}(\mathbf{Set}) \to [\mathbf{Set}, \mathbf{Set}]$, meaning that directed containers are the same as containers whose interpretation carries a comonad structure.

[1] This is not the same as having containers with suitable additional structure interpret into monads under the standard interpretation of containers into set functors.

Here we are interested in the "cointerpretation" of containers given by

$$\langle\!\langle S, P \rangle\!\rangle^{\mathrm{c}}\, X = \Pi s : S.\, P s \times X \cong (\Pi s : S.\, P\, s) \times (S \to X)$$

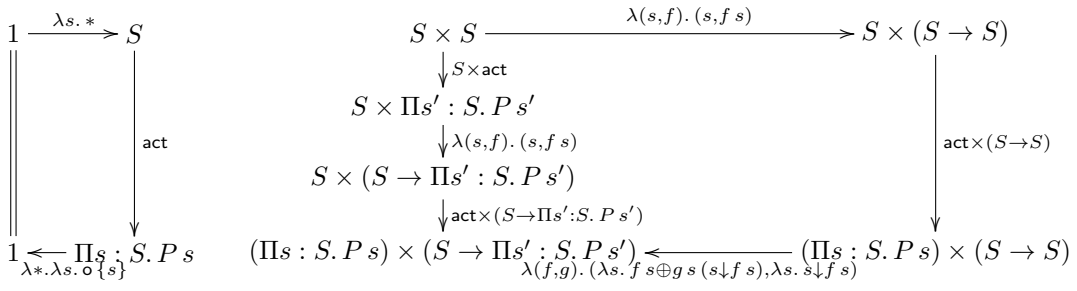
The functor $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}} : \mathbf{Cont}^{\mathrm{op}} \to [\mathbf{Set}, \mathbf{Set}]$ fails to be monoidal (for the monoidal structure on $\mathbf{Cont}^{\mathrm{op}}$ taken from $\mathbf{Cont}$), but it is lax monoidal. It is neither full nor faithful.

It is straightforward that $\mathbf{DCont}^{\mathrm{op}} \cong (\mathbf{Comonoids}(\mathbf{Cont}))^{\mathrm{op}} \cong \mathbf{Monoids}(\mathbf{Cont}^{\mathrm{op}})$. It follows therefore that directed containers cointerpret to monads via

$$\begin{aligned}
&\eta : \forall \{X\}.\, X \to \Pi s : S.\, P\, s \times X \\
&\eta\, x = \lambda s.\, (\mathsf{o}\, \{s\}, x) \\
&\mu : \forall \{X\}.\, (\Pi s : S.\, P\, s \times \Pi s' : S.\, P\, s' \times X) \to \Pi s : S.\, P\, s \times X \\
&\mu\, f = \lambda s.\, \mathsf{let}\ (p, g) = f\, s;\ (p', x) = g\, (s \downarrow p)\ \mathsf{in}\ (p \oplus p', x)
\end{aligned}$$

i.e., $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}}$ extends to a functor $\langle\!\langle - \rangle\!\rangle^{\mathrm{dc}} : \mathbf{DCont}^{\mathrm{op}} \to \mathbf{Monads}(\mathbf{Set})$. We do not get that the functor $\langle\!\langle - \rangle\!\rangle^{\mathrm{dc}}$ is the pullback of $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}}$ along $U : \mathbf{Monads}(\mathbf{Set}) \to [\mathbf{Set}, \mathbf{Set}]$.

$\langle\!\langle - \rangle\!\rangle^{\mathrm{dc}}$ describes the free models of the (generally non-finitary) Lawvere theory given by one operation $\mathsf{act} : S \to \Pi s : S.\, P\, s$ and two equations



For cointerpretation, it is useful to think of elements of S as states, those of $P\, s$ as updates applicable to a state s, $s \downarrow p$ as the result of applying an update p to the state s, $\mathsf{o}\{s\}$ as the nil update, $p \oplus p'$ composition of two updates. Monads induced by directed containers generalize the state monad much in the spirit of the generalizations considered by Kammar [2], but still a bit differently. The state monad $TX = S \to S \times X$ is recovered by taking $S = S$, $P\, s = S$, $s \downarrow p = p$, $\mathsf{o}\, \{s\} = s$, $p \oplus p' = p'$. The directed container for the nonempty list comonad, $S = \mathsf{Nat}$, $P\, s = [0..s]$, $s \downarrow p = s - p$, $\mathsf{o} = 0$, $p \oplus p' = p + p'$ gives us a monad on the functor $TX = \Pi s : \mathsf{Nat}.\, [0..s] \times X$. The states are natural numbers; the updates applicable to a state s are numbers not greater than s; applying an update means decrementing the state.

Acknowledgements We thank Ohad Kammar for discussions. This ongoing work is being supported by the University of Edinburgh Principal's Career Development PhD Scholarship, the ERDF funded Estonian Centre of Excellence in Computer Science, EXCS, and the Estonian Research Council grant no. 9475.

References

[1] D. Ahman, J. Chapman, T. Uustalu. When is a container a comonad? In L. Birkedal, ed., *Proc. of 15th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2012 (Tallinn, March 2012)*, v. 7213 of *Lect. Notes in Comput. Sci.*, pp. 74–88. Springer, 2012.

[2] O. Kammar. Take action for your state: effective conservative restrictions. Slides from Scottish Programming Language Seminar, Strathclyde, 2010.

Mendler-style Recursion Schemes for Mixed-Variant Datatypes

Ki Yung Ahn[1], Tim Sheard[1] and Marcelo Fiore[2]

[1] Portland State University *
[2] University of Cambridge

The context of our work is the Nax project. Our goal is to develop a language system, called Nax, which supports the merits of both functional programming languages and formal reasoning systems based on the Curry–Howard correspondence. Our approach towards these goals is to design an appropriate foundational calculus [3] that extends F_ω[5] (or Fix_ω[1] similarly) to justify the theory of *Mendler-style recursion schemes* [6] with *term-indexed datatypes*.

In this abstract, we outline a paper that will ① discuss the advantages of the Mendler style, ② report that we can define an evaluator for the simply-typed HOAS using Mendler-style iteration with syntactic inverses (`msfit`), and ③ propose a new recursion scheme (work in progress) whose termination relies on the invariants specified by size measures on indices.

Advantages of the Mendler style include allowing arbitrary definition of recursive datatypes, while still ensuring well-behaved use by providing a rich set of principled eliminators. Certain concepts, such as HOAS, are most succinctly defined as mixed-variant datatypes, which are unfortunately, outlawed in many existing reasoning systems (e.g., Coq, Agda). One is forced to devise clever encodings [4], to use concepts like HOAS within such systems.

In functional *programming* languages, for instance, in Haskell, a HOAS for the untyped λ-calculus can be defined as `data Exp = Abs (Exp -> Exp) | App Exp Exp`. Even if we assume all functions embedded in `Abs` are non-recursive, evaluating HOAS may still cause problems for logical reasoning, since the untyped λ-calculus has diverging terms. However, there are many well-behaved (i.e., terminating) computations on `Exp`, such as converting an HOAS expression to first-order syntax. Ahn and Sheard [2] formalized a Mendler-style recursion scheme (`msfit`, a.k.a. *msfcata*) that captures these well-behaved computations.

If the datatype `Exp` had indexes to assert invariants of well-formed expressions, we could rely on these invariants to write even more expressive programs, such as a terminating well-typed evaluator. Discussion around this idea will constitute the latter parts of the paper.

A simply-typed HOAS evaluator can be defined using `msfit` at kind `* -> *`. Since `msfit` terminates for any datatype, we are also proving that the evaluation of the simply-typed λ-calculus always terminates just by defining `eval : Exp t -> Id t` in Nax, as below. We wonder `eval` has similarities to other normalization strategies like NbE [7].

```
data E : (* -> *) -> (* -> *) where      -- the "deriving fixpoint Exp" defines
  Abs : (r a -> r b) -> E r (a -> b)     --   abs f   = In[* -> *] (Abs f)
  App : E r (a -> b) -> E r a -> E r b   --   app f e = In[* -> *] (App f e)
    deriving fixpoint Exp                --   synonym Exp t = Mu[* -> *] E t

data Id a = MkId a -- the identity type
unId (MkId x) = x  -- destructor of Id

eval e = msfit { t . Id t } e with
           call inv (App f x) = MkId (unId(call f) (unId(call x)))
           call inv (Abs f) = MkId (\v -> unId(call (f (inv (MkId v)))))
```

*supported by NSF grant 0910500.

The type of `eval : Exp t -> Id t` is inferred from { `t . Id t` }, which specifies the answer type in relation to the input type's index. Conceptually, `msfit` at kind `* -> *` has the following type.

```
msfit : (forall r . (forall i . r i -> ans i) -- call
                 -> (forall i . ans i -> r i) -- inv
                 -> (forall i . f r i -> ans i)      ) -> Mu[* -> *] f j -> ans j
```

Evaluation via user-defined value domain, instead of the native value space of Nax, motivates a new recursion scheme, `mprsi`, standing for Mendler-style primitive recursion with sized index. Consider writing an evaluator `veval : Exp t -> Val t` via the value domain `Val : * -> *`.

```
data V : (* -> *) -> * -> * where      -- the "deriving fixpoint Val" defines
  Fun : (r a -> r b) -> V r (a -> b)   -- fun f = In [* -> *] (Fun f)
    deriving fixpoint Val              -- synonym Val t = Mu[* -> *] V t

veval e = msfit { t . V t } e with
            call inv (App f x) = unfun (call f) (call x)  -- how do we define unfun?
            call inv (Abs f) = fun (\v -> (call (f (inv v))))
```

Only if we were able to define `unfun : Val (a -> b) -> Val a -> Val b`, this would be admitted in Nax. However, it is not likely that `unfun` can be defined using recursion schemes currently available in Nax. Thereby, we propose a new recursion scheme `mprsi`, which extends the Mendler-style primitive recursion (`mpr`) with the uncasting operation.

```
mprsi : (forall r . (forall i. r i -> ans i)                       -- call
                 -> (forall i. (i < j) => r i -> Mu[* -> *] f i)   -- cast
                 -> (forall i. (i < j) => Mu[* -> *] f i -> r i)   -- uncast
                 -> (forall i. f r i -> ans i) ) -> Mu[* -> *] f j -> ans j

unfun v = mprsi { (a -> b) . V a -> V b } v with
            call cast (Fun f) = cast . f . uncast   -- dot (.) is function composition
```

Note the size constraints `(i < j)` on both `cast` and `uncast` operations (FYI, `mpr`'s `cast` does not have size constraints). These constraints prevent writing evaluator-like functions on strange expressions that have constructors like below, which may cause non-termination.

```
app1 : (Exp1 (a->b) -> Exp1 b) -> Exp1 (a->b)  -- prevented by constraint on uncast
app2 : (Exp2 a -> Exp2 (a->b)) -> Exp2 (a->b)  -- prevented by constraint on cast
```

Our examples in this abstract only involve type indices, but similar formulation is possible for term indices as well. This is still a fresh proposal awaiting proof of termination.

References

[1] Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In *CSL*, volume 3210 of *LNCS*, pages 190–204. Springer, 2004.

[2] Ki Yung Ahn and Tim Sheard. A hierarchy of Mendler-style recursion combinators: Taming inductive datatypes with negative occurrences. In *ICFP '11*, pages 234–246. ACM, 2011.

[3] Ki Yung Ahn, Tim Sheard, Marcelo Fiore, and Andrew M. Pitts. System Fi: a higher-order polymorphic lambda calculus with erasable term indices. In *TYPES*, LICS, 2013. (to appear).

[4] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP '08*, pages 143–156. ACM, 2008.

[5] Girard, J.-Y. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972.

[6] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987.

[7] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 203–211, 1991.

Univalent categories and the Rezk completion

Benedikt Ahrens[1], Krzysztof Kapulkin[2] and Michael Shulman[1]

[1] Institute for Advanced Study, Princeton, NJ, USA
ahrens@ias.edu, mshulman@ias.edu
[2] University of Pittsburgh, Pittsburgh, PA, USA
krk56@pitt.edu

When formalizing category theory in traditional, set-theoretic foundations, a significant discrepancy between the foundational notion of "sameness"—*equality*—and its categorical notion arises: most category–theoretic concepts are invariant under weaker notions of sameness than equality, namely isomorphism in a category or equivalence of categories. We show that this discrepancy can be avoided when formalizing category theory in Univalent Foundations.

The *Univalent Foundations* is an extension of Martin-Löf Type Theory (MLTT) recently proposed by V. Voevodsky [4]. Its novelty is the *Univalence Axiom* (UA) which closes an unfortunate incompleteness of MLTT by providing "more equalities between types". This is obtained by identifying equality of types with equivalence of types. To prove two types equal, it thus suffices to construct an equivalence between them.

When formalizing category theory in the Univalent Foundations, the idea of Univalence carries over. We define a *precategory* to be given by a type of objects and, for each pair (x, y) of objects, a *set* $\hom(x, y)$ of morphisms, together with identity and composition operations, subject to the usual axioms. In the Univalent Foundations, a type X is called a *set* if it satisfies the principle of Uniqueness of Identity Proofs, that is, for any $x, y : X$ and $p, q : \mathsf{Id}(x, y)$, the type $\mathsf{Id}(p, q)$ is inhabited. This requirement avoids the introduction of coherence axioms for associativity and unitality of categories.

A *univalent* category is then defined to be a category where the type of isomorphisms between any pair of objects is equivalent to the identity type between them. We develop the basic theory of such univalent categories: functors, natural transformations, adjunctions, equivalences, and the Yoneda lemma.

Two categories are called *equivalent* if there is a pair of adjoint functors between them for which the unit and counit are natural isomorphisms. Given two categories, one may ask whether they are equal in the type-theoretic sense—that is, if there is an identity term between them in the type of categories—or whether they are equivalent. One of our main results states that for univalent categories, the notion of (type-theoretic) *equality* and (category-theoretic) *equivalence coincide*. This implies that properties of univalent categories are automatically invariant under equivalence of categories—an important difference to the classical notion of categories in set theory, where this invariance does not hold.

Moreover, we show that any category is weakly equivalent to a univalent category—its *Rezk completion*—in a universal way. It can be considered as a truncated version of the Rezk completion for Segal spaces [3]. The Rezk completion of a category is constructed via the Yoneda embedding of a category into its presheaf category, a construction analogous to the *strictification* of bicategories by the Yoneda embedding into Cat, the 2-category of categories.

Large parts of this development have been formally verified [1] in the proof assistant Coq, building on Voevodsky's *Foundations* library [5]. In particular, the formalization includes the Rezk completion together with its universal property.

A preprint covering the content of this talk is available on the arXiv [2].

Acknowledgments The authors thank Vladimir Voevodsky for many helpful conversations.

This material is based upon work supported by the National Science Foundation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Rezk completion, formalized. `https://github.com/benediktahrens/rezk_completion`.

[2] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. arXiv:1303.0584, 2013.

[3] Charles Rezk. A model for the homotopy theory of homotopy theory. *Trans. Amer. Math. Soc.*, 353(3):973–1007 (electronic), 2001.

[4] Vladimir Voevodsky. Univalent foundations project. `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/univalent_foundations_project.pdf`.

[5] Vladimir Voevodsky. Univalent foundations repository. ongoing `Coq` development.

Formal Non-linear Optimization via Templates and Sum-of-Squares

Xavier Allamigeon[1], Stéphane Gaubert[2], Victor Magron[3] and Benjamin Werner[4*]

[1] INRIA and CMAP,
École Polytechnique, Palaiseau, France,
Xavier.Allamigeon@inria.fr
[2] INRIA and CMAP,
École Polytechnique, Palaiseau, France,
Stephane.Gaubert@inria.fr
[3] INRIA and LIX,
École Polytechnique, Palaiseau, France,
magron@lix.polytechnique.fr
[4] INRIA and LIX,
École Polytechnique, Palaiseau, France,
benjamin.werner@polytechnique.edu

Formalization of mathematics involve challenging problems as computing a certified lower bound of a real-valued multivariate function f in a compact semialgebraic set K. The goal of global optimization is to find the global minimum $f^* := \inf_{x \in K} f(x)$ and a global minimizer x^*. Optimization tools may provide certificates to verify that a given lower bound of f^* is valid.

Many theorems can be proven by solving such problems, as the thousands of non-linear inequalities issued from the Flyspeck project [3] (formal proof of Kepler Conjecture by Thomas Hales). It requires to interface a global optimization framework with a proof assistant such as Coq. Recent efforts have been made to perform a formal verification of these inequalities with Taylor interval approximations in HOL-Light [9].

Formal methods that produce precise bounds are mandatory to complete the Flyspeck project since the inequalities are in general tight, and thus challenging for numerical solvers. Furthermore, such methods should also tackle scalability issues, which arise when one wants to provide coarse lower bounds for large-scale optimization problems. The scalability depends on how much grows the number of polynomial inequalities and the system variables.

The proofs assistants have a limited computing power, hence our aim is to devise algorithms that produce a concise certificate for each optimization problem and ensure that checking this certificate is computationally reasonably simple.

Such algorithms rely on hybrid symbolic-numeric certification methods (see e.g. Kaltofen, Peyrl and Parrilo [5]). It allows to produce positivity certificates which can be checked in proof assistants such as Coq [6] [2], HOL-light [4] or MetiTarski [1]

Our function f is constituted of semialgebraic operations as well as univariate transcendental functions. We bound some of these constituents by suprema or infima of quadratic forms (max-plus approximation method). The initial problem is relaxed into semialgebraic optimization problems which we solve by sparse semidefinite programming (SDP) relaxations (SparsePOP solver by Kojima [8]). The size of these relaxations can be controlled by an adaptive semialgebraic approximations scheme that generalizes the linear templates method by Manna et al. [7]. Moreover, we outline the conversion of the numerical Sum-of-Squares produced by the SDP

*The research leading to these results has received funding from the European Union's 7^{th} Framework Programme under grant agreement nr. 243847 (ForMath).

solvers into an exact rational certificate. We explain how to implement the method in Coq in order to formally check the validity of the certificate.

We illustrate the efficiency of our framework with various examples from the global optimization literature, as well as inequalities issued from the Flyspeck project.

References

[1] Behzad Akbarpour and Lawrence Charles Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *J. Autom. Reason.*, 44(3):175–205, March 2010.

[2] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Proceedings of the 2006 international conference on Types for proofs and programs*, TYPES'06, pages 48–62, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.

[4] John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider and Jens Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118, Kaiserslautern, Germany, 2007. Springer-Verlag.

[5] Erich L. Kaltofen, Bin Li, Zhengfeng Yang, and Lihong Zhi. Exact certification in global polynomial optimization via sums-of-squares of rational functions with rational coefficients. *JSC*, 47(1):1–15, jan 2012. In memory of Wenda Wu (1929–2009).

[6] David Monniaux and Pierre Corbineau. On the generation of Positivstellensatz witnesses in degenerate cases. In Marko Van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP)*, volume 6898 of *Lecture Notes in Computer Science*, pages 249–264. Springer Verlag, August 2011.

[7] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In Radhia Cousot, editor, *Proc. of Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385, pages 21–47, Paris, France, January 2005. Springer Verlag.

[8] Hayato Waki, Sunyoung Kim, Masakazu Kojima, and Masakazu Muramatsu. Sums of squares and semidefinite programming relaxations for polynomial optimization problems with structured sparsity. *SIAM Journal on Optimization*, 17:218–242, 2006.

[9] Alexey Solovyev and Thomas C. Hales. Formal verification of nonlinear inequalities with Taylor interval approximations. *CoRR*, abs/1301.1702, 2013.

Weak ω-groupoids and beyond. (Abstract)

Thorsten Altenkirch

School of Computer Science
University of Nottingham

Our goal is to develop an ω-groupoid model of Type Theory to justify Homotopy Type Theory, including principles like functional extensionality, propositional extensionality (equivalent predicates are equal) and univalence (isomorphic sets are equal). A first step here was [AR12] which set up a syntactic framework for describing what is a weak ω-groupoid. This definition was recently very much simplilfied by Guillaume Brunerie [Bru12] based on a suggestion by Grothendieck. We show that using Brunerie's approach we can formally show that identity types give rise to a weak ω-groupoid this internalizing [BG08, Lum09]. Going further we discuss the notion of a weak morphism between weak ω groupoids.

References

[AR12] Thorsten Altenkirch and Ondrej Rypacek. A syntactical approach to weak omega-groupoids. In *CSL*, pages 16–30, 2012.

[BG08] Benno Van Den Berg and Richard Garner. Types are weak ω-groupoids, 2008.

[Bru12] Guillaume Brunerie. Syntactic Grothendieck weak ω-groupoids. Available from the HoTT wiki, February 2012.

[Lum09] Peter Lumsdaine. Weak ω-categories from intensional type theory. *Typed lambda calculi and applications*, pages 172–187, 2009.

Strong Normalization for Intuitionistic Arithmetic with 1-Excluded Middle

Federico Aschieri[1], Stefano Berardi[2] and Giovanni Birolo[2]

[1] Équipe Plume, LIP, École Normale Supérieure de Lyon, France
[2] Università di Torino, Italy

We present a new set of reductions for derivations in natural deduction or type-theoretical form, that can extract witnesses from closed derivations of simply existential formulas in Heyting Arithmetic (HA) plus the Excluded Middle Law, restricted to simply existential formulas (EM1). EM1 is the axiom schema $\forall x.P(x,y) \vee \exists y.\neg P(x,y)$, for any primitive recursive binary predicate P: we refer to [2] for a presentation of HA and EM1. The interest of EM1 lies in the fact that this classical principle is logically simple, yet it may formalize many classical proofs: for instance, proofs of Euclidean geometry (like Sylvester conjecture, see J. von Plato [14]), of Algebra (like Dickson's Lemma, see S. Berardi [7]) and of Analysis (those using König's Lemma, see Kohlenbach [10]). Our rule for EM1 is just an $\vee$-elimination: for any primitive recursive P, we deduce C from $\forall y.P(x,y) \vdash C$ and $\exists y.\neg P(x,y) \vdash C$.

The reductions we have for classical logic are inspired by the informal idea of learning by making falsifiable hypothesis. The informal idea expressed by our reductions is that whenever we fix some value $x = n$ in the EM1-rule, we assume $\forall y.P(n,y)$ and we try to produce some proof of C. Whenever we need an instance $P(n,m)$ of the assumption $\forall y.P(n,y)$ we check it, and if it is true we replace it by its canonical proof. If all instances $P(n,m)$ we checked of $\forall y.P(n,y)$ are true, *and no assumption* $\forall y.P(n,y)$ *is left* (this is the non-trivial part), then the proof of C is independent from $\forall y.P(n,y)$ and we succeed in proving C. Remark that, in this case, we do not know whether $\forall y.P(n,y)$ is true or false, because we only checked finitely many instances of it: all we do know is that $\forall y.P(n,y)$ is unnecessary. If instead some assumption of $\forall y.P(n,y)$ is left we are stuck. If at any moment we need some instance $P(n,m)$ which is false, then we falsified the assumption $\forall y.P(n,y)$. In this case the attempt of proving C from $\forall y.P(n,y)$ fails, we obtain $\neg P(n,m)$ and we *raise an exception* (using a terminology from functional languages) and we continue using the proof of $\exists y.\neg P(n,y)$. From the knowledge that $\neg P(n,m)$ holds, we may provide a canonical proof of $\exists y.\neg P(n,y)$. Therefore we get a proof of C and we terminate. We have two results for our formal set of rules: a *normal form result*, expressing that any normal proof of a simply existential C provides a witness of C through the process sketched above (that is, we never get stuck if C is simply existential); and a *strong normalization result*, proving that all reduction paths terminate into a normal form.

Unlike Interactive realizability [4], our reductions are in the research line starting from Goedel $\neg\neg$-translation, Friedman's A-translation, Griffin's continuations, and continuing with Parigot's [13] $\lambda\mu$-calculus , Krivine Classical realizability [11], Herbelin's symmetry between call-by-name and call-by-value [1], the works on classical realizability by Miquel [12] and others.

There also are differences, however. For example, in all interpretations of classical logic coming from Friedman's A-translation, the constant $\bot$ (false) is treated as a variable and eventually replaced by the simply existential formula C which is the goal of the proof. We do not replace $\bot$, and we provide instead a constructive interpretation of the classical axiom EM1 in term of learning (see [6]).

Another difference is that we extract a functional program using *delimited exceptions* and we do not have explicit continuations, as it is the case with the other interpretations. We raise an exception when we find an instance $P(n,m)$ of the assumption $\forall y.P(n,x)$ which is

false, but in this case we do not restart the whole program, we only restart the part of it relying on $\forall y.P(n,x)$. Indeed, we switch from the execution of the left assumption of one rule EM1 to the execution of the right assumption of the same rule. There are more technical differences. First of all, our reduction set is non-deterministic. Whenever there are two false instances $P(n,m)$, $P(n,m')$ of an assumption $\forall y.P(n,y)$ in some EM1-rule, we may either raise the exception related to $P(n,m)$, or raise the exception related to $P(n,m')$. The computation is converging in both cases, and the witness we get for a simple existential conclusion C is correct in both cases: however, we may obtain a *different* witness in the two cases. Second, as we said our exception is delimited, therefore sometimes we have the problem of extending the delimitation of the exception the minimum necessary to avoid getting stuck. Griffin solves this problem using unlimited exceptions and continuations. Instead, we prefer having the smallest possible delimitation for our exceptions, and we solve the problem using Prawitz's permutative reductions for disjunction elimination rules, which have the task of duplicating the context.

We consider our work as an application of some of the ideas of Interactive realizability [4] to the research line based on control operators. This approach leads to a different realizability semantics from Krivine's (but also from Interactive Realizability). This work should also be useful as a first step toward a possible implementation of Interactive realizability (another more expressive one can be found in Aschieri [5]).

The strong normalization proof uses a combination of realizability and a technique introduced by Aschieri-Zorzi [5], consisting in studying normalization in a system extended with de' Liguoro-Piperno non-deterministic choice operator [9]. The proof of the normal form property uses an extended notion of main branch developed in the ph.d. thesis of G. Birolo [8].

References

[1] Z. M. Ariola, H. Herbelin, A. Saurin: Classical Call-by-Need and Duality. TLCA 2011: 27-44

[2] Y. Akama, S. Berardi, S. Hayashi, U. Kohlenbach, An Arithmetical Hierarchy of the Law of Excluded Middle and Related Principles. LICS 2004, pages 192-201.

[3] F. Aschieri, Interactive Realizability for Second-Order Heyting Arithmetic with $EM1$ and $SK1$, Technical Report, `http://hal.inria.fr/hal-00657054`.

[4] F. Aschieri, S. Berardi, Interactive Learning-Based Realizability for Heyting Arithmetic with EM1, Logical Methods in Computer Science, 2010

[5] F. Aschieri, M. Zorzi, Non-determinism and the Strong Normalization of System T, manuscript.

[6] F. Aschieri, S. Berardi, *A New Use of Friedman's Translation: Interactive Realizability*, in: Logic, Construction, Computation, Berger et al. eds, Ontos-Verlag Series in Mathematical Logic, 2012.

[7] Stefano Berardi: Some intuitionistic equivalents of classical principles for degree 2 formulas. Ann. Pure Appl. Logic 139(1-3): 185-200 (2006)

[8] Giovanni Birolo: Interactive Realizability, Monads and Witness Extraction, Ph. d. thesis, April, 1 2013, Università di Torino.

[9] Ugo de'Liguoro, Adolfo Piperno: Non Deterministic Extensions of Untyped Lambda-Calculus. Inf. Comput. 122(2): 149-177 (1995)

[10] Ulrich Kohlenbach: On uniform weak König's lemma. Ann. Pure Appl. Logic 114(1-3): (2002)

[11] Jean-Louis Krivine: Realizability in Classical Logic. Panoramas et Syntheses 27 (2009), 197-229

[12] Alexandre Miquel: Existential witness extraction in classical realizability and via a negative translation. Logical Methods in Computer Science 7(2) (2011)

[13] Michel Parigot: Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. LPAR 1992: 190-201

[14] Jan von Plato: A Constructive Approach to Sylvester's Conjecture. J. UCS 11(12): 2165-2178 (2005)

Towards Infinite Terms in Twelf

Maxime Beauquier

IT-Universitetet i København,
maxime.beauquier@itu.dk

Representing and reasoning about infinite terms is a great challenge in mathematics [1,3,4,7], and in particular in computer science [2, 6]. Infinite terms are widely used in both theoretical (sets, reals, graphs, Büchi automata, rewriting systems, process calculi, ...) and applied (traces of computations, servers, streams of data, parallel executions, ...) computer science.

Our interest is in mechanised proofs, i.e. theorem provers and proof assistants. These raise questions of the representation of theories – finite or not – that involve infinite data structures. There are several mathematical formalisms for finite representation of infinite terms like transfinite induction, ordinal assignment, bar induction, coinduction and others, which have been used to build mechanised proofs on infinite data-structure or infinite behaviour; these systems are complex and it is hard to reason about these systems.

In this paper, we model infinite terms with anamorphisms, morphisms of the form $A_0 \vee \ldots \vee A_n \to A$[1]. More precisely, infinite terms are represented by their *unfolding*, which can also be seen as their computation, behaviour, or more generally *observation*. Every infinite terms must enjoy a progress property called *productivity*: it is always possible to unfold – or observe – an infinite term.

Twelf is a proof assistant, based on the logical framework LF [5], that supports dependent types and higher-order abstract syntax with canonical terms. In Twelf, LF judgements are represented as types with a stratified syntax of kinds, type families and terms. Twelf derives its induction principles on the inductive definition of canonical forms of LF terms, and by extension on LF type derivations. Well-formedness of an LF relation $R : A_0 \to \ldots \to A_n \to$ `type` is decided by a totality property (separated into two properties: coverage and termination), which states that, according to a mode given to the arguments of R[2], for all input term $t : A_0 \times \ldots \times A_m$ there exists a output term $u : A_{m+1} \times \ldots \times A_n$ such that $R(t, u)$, where $(m < n)$, and this process terminates.

Based on the work of Noam Zeilberger on defunctionalisation [9], we have built an encoding method that allows us to describe adequately, within Twelf (i.e. using induction as the reasoning principle), cyclic behaviours of some infinite terms in a finite number of observations and reason about it in Twelf. The *apply* function given by the defunctionalisation is here an *observe* function for the infinite term; the productivity of the *observe* function is ensure by Twelf totality checker.

The encoding can be summarised as: from an anamorphism $A_0 \vee \ldots \vee A_n \to A$, the disjunction $A_0 \vee \ldots \vee A_n$ is represented by an LF type, namely α, which is an inductive structure for the observations of A. Another LF type, namely α`*`, represents the set of defined infinite terms, i.e. terms of type A. At last, a Twelf relation, ε `:  α* -> α -> type`, interpreted as $\forall$`t:`α`*.`$\exists$`u:`α`.`ε `t u`[3], acts as an observation function. The Twelf logic programming paradigm is used to ensure that this observation function is a terminating function that maps any representation of infinite terms to an observation. The coverage and termination properties and type checking of the observation function ensure productivity.

[1] Anamorphisms are the dual of catamorphisms (morphisms of the form $A_0 \wedge \ldots \wedge A_n \to A$, which can be used to represent inductive definitions.

[2] i.e. if they are an input or an output of R.

[3] i.e. moded as α^* is the input and α is the output.

We have running examples of representation of some infinite terms, such as conats, colists, streams or infinite typed λ-calculus. We show here a small example of an implementation of data streams in Twelf, such as the representation of the stream of ones, or the LF function which merges two streams. The observation relation is checked to be total, i.e. it covers all the cases of the inputs (`stream*`) and is terminating.

```
stream* : type.
stream : type.
cons : nat -> stream* -> stream.

ε/stream : stream* -> stream -> type.
%mode ε/stream +D -D'.

ones : stream*.
ε/ones : ε/stream ones (cons (s z) ones).

merge : stream* -> stream* -> stream*.
ε/merge : ε/stream (merge S T) (cons N (merge T S'))
           <- ε/stream S (cons N S').

%worlds () (ε/stream _ _).
%total (S) (ε/stream S _).
```

In this runnable example, the observation function `ε/stream` is checked to be a total function over `stream*`, which induces productivity of all infinite terms represented by `ones` and `merge(S,S')` (where `S` and `S'` are themselves either `ones` or `merge(T,T')`).

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 27–38. ACM, 2013.

[2] AgdaWiki. `http://wiki.portal.chalmers.se/agda/agda.php`.

[3] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-Oblivious Stream Productivity. In *Proc. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2008)*, number 5330 in LNCS, pages 79–96. Springer, 2008.

[4] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *TYPES*, pages 39–59, 1994.

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

[6] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[7] G.E. Mints. Finite investigations of transfinite derivations. *Journal of Soviet Mathematics*, 10:548–596, 1978.

[8] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[9] Noam Zeilberger. Defunctionalizing focusing proofs, 2009. In International Workshop on Proof-Search in Type Theories 2009.

A learning-based interpretation for polymorphic λ-calculus *(ongoing work)*

Stefano Berardi

Universitá di Torino,
Corso Svizzera 185, 10149 Torino

Abstract

We propose a model of polymorphic λ-calculus or system F. Types and terms are interpreted as well-founded trees. A map using a type as input is considered to be correct if it maps well-founded trees into well-founded trees. We define a map over types over a "test set" of previously defined types, in such a way that the correctness of the map over the test set implies the correct of the map over all types, including those which are not yet defined. The construction is a modification of the idea of Dilator due to Jean-Yves Girard [3], [4].

In system F [1], [2], a type $\forall\alpha.A$ is defined by making reference to all types $A[T/\alpha]$, for any type T, including the very type $T = \forall\alpha.A$ we are defining, and which is not yet available for the definition. We interpret the definition of a type $\forall\alpha.A$ as a construction able to "learn" how to process new inputs: it has an index set which is "open" to new individual in the sense of Martin-Löf, which may be extended to include any type we will define later, including the very type we are defining.

We represent both types and terms of F by well-founded trees, possibly indexed over the set itself of all well-founded trees of the model. In the last case we have to make precise which kind of operations are allowed over a "non-yet defined tree", that is, how we may "learn" how to process new kinds of inputs. In a predicative interpretation, when we have a tree $T = \bigvee^L_{i\in I} T_i$ of label L as input, we usually do not know how the index function $i \mapsto T_i$ of the tree is defined, but we do know the index set I of the tree. In this case we first read the label L, then we use the tree as a kind of "black box", having an input channel ending in the index set I, and an output channel returning some immediate subtree T_i of T. For finitely many times we insert an index $i \in I$ in the input channel, and we obtain the corresponding subtree T_i in the output channel. Now assume that we have as input a tree $T = \bigvee_{i\in I} T_i$ whose index set I is *not yet known*: for instance, I is the set of all well-founded trees of the model, which will be defined using the very tree we are building now. In this case the input channel of the tree is not available to us: we cannot select an index $i \in I$. The only way of using T as input is to produce, as output, a tree $U = \bigvee_{i\in I} U_i$ with *the same index set* as T. In this way we ask to the "external world" for some index i in the unknown index set I. When we will know how to use the input channel ending in I, we will send some $i \in I$ through it. In this moment, we will learn how to use the channel I, and we use $i \in I$ to select the subtree T_i in the input tree. Then we compute the output U_i out of the value T_i. We call this way of computing *"polymorphism"*, because it works for *many possible shapes* of the unknown elements of the index set I, instead of working for an index set whose elements are known. To be accurate we speak of level 1 polymorphism or 1-polymorphism. We call known sets *"monomorphic"*, because their elements have a unique possible shape.

We call 1-polymorphic the trees having, for all uncountable branching, an index map which is 1-polymorphic. We use 1-polymorphic trees to interpret the fragment of F with second order quantifiers not depending on each other. The main feature of a 1-polymorphic tree $T = \bigvee_{i\in I} T_i$, with I uncountable, is that we may extend the index set I of T to the set of all well-founded

trees, including the tree T itself. Apparently, this circularity prevents any possibility of well-foundation and therefore of correctness of the construction, but this is not the case. The reason is that we may restrict these branching to some "threshold" set I_0, including enough trees to decide the correctness of the construction. There is a similarity with what we do in a Scott domain $D = [D \to D]$, when we restrict a continuous map $f \in [D \to D]$ to all "finite" inputs. Thanks to continuity, we deduce the behavior of f over all $d \in D$, including f itself, because $D = [D \to D]$. In the case of 1-polymorphic maps, however, the threshold set is not used to infer the behavior of f but to infer the correctness of f, and therefore it is larger than the threshold set for Scott continuous functions. We will prove that the "threshold set" for 1-polymorphic trees is the set Ω_1 of well-founded trees with at most countable branchings.

We consider a 1-polymorphic tree T to denote a correct construction if by restricting all uncountable branchings of T to the "threshold set" we obtain some well-founded pruning of T. One may object that if we extend the index set to all trees, we obtain a tree $T = \bigvee\{T_i | i \text{ tree}\}$ which may be instanced to any tree i, not just to the trees i belonging to the "threshold set" Ω_1. How can we guarantee that *any* instance T_i of T is correct (that is, well-founded), including the instance $i = T$? We will use a similarity between our construction and Girard's Dilators, and we will prove that the output of a 1-polymorphic map is well-founded if all its countable subsets are well-founded. We will prove that each countable subsets of the output depends on some countable subset of the input, and therefore that any 1-polymorphic map sending countable well-founded trees into countable well-founded trees sends well-founded trees into well-founded trees. This result extends the input map $i \mapsto T_i$ from the threshold set Ω_1 to all well-founded trees, while preserving the fact that the output is well-founded.

We denote the set of 1-polymorphic trees with Ω_2. It is possible to nest this construction $2, 3, \ldots$ times, and to define level $2, 3, \ldots$ polymorphic 2-polymorphic maps, somehow corresponding to Girard's ptykes. For any $k \in N$, k-polymorphic trees may be used to interpret the fragment of F with k nesting of quantifiers. The union of all these constructions is a model of the whole F.

References

[1] Stefano Berardi, Chantal Berline: Building continuous webbed models for system F. Theor. Comput. Sci. 315(1): 3-34 (2004)

[2] Stefano Berardi, Makoto Tatsuta: Internal models of system F for decompilation. Theor. Comput. Sci. 435: 3-20 (2012)

[3] Jean-Yves Girard: Π^1_2-logic, Part 1: Dilators. Annals of Mathematical Logic. Volume 21, Issues 23, December 1981, Pages 75219

[4] Jean-Yves Girard: The System F of Variable Types, Fifteen Years Later. Theor. Comput. Sci. 45(2): 159-192 (1986)

Are streamless sets Noetherian? (Work in progress)

Marc Bezem[1], Thierry Coquand[2] and Keiko Nakata[3]

[1] Department of Informatics, University of Bergen
[2] Department of Computing Science, Chalmers University
[3] Institute of Cybernetics at Tallinn University of Technology

Abstract

In our previous works, we studied a notion of finiteness from a viewpoint of constructive mathematics. Constructively, there are at least four positive definitions for a set of natural numbers being finite, two of which are of our interest: a Noetherian set and a streamless set. A set is Noetherian if the root of the tree of duplicate-free lists over the set is inductively accessible. A set is streamless if every stream over the set has a duplicate. In this talk, I will present our on-going work, aiming at positively answering to our conjecture: it is constructively unprovable that a streamless set is Noetherian.

Classically any set is either finite or infinite, and any finite set is bijective to the set $\{n \mid n \leq m\}$ of natural numbers n that are less than m for some m. However, the picture becomes much more subtle in a constructive setting. In our previous works [2, 1], we studied, from a constructive point of view, several classically equivalent definitions of finiteness, in terms of their relative strength and closure properties.

Constructively, there are at least four positive definitions for finiteness. Assuming that the set A in question has decidable equality, the four variations may be stated as follows.

(i) The set A is given by a list. (Enumerated sets)

(ii) There exists a bound such that any list over A contains duplicates whenever its length exceeds the bound. (Size-bounded sets)

(iii) The root of the tree of duplicate-free lists over A is inductively accessible. (Noetherian sets)

(iv) Every stream over A has a duplicate. (Streamless sets)

These four notions are classically equivalent but of decreasing constructive strength. Indeed, collapsing any two of the first three variations gives rise to classical power: for instance, from (i) $\Leftrightarrow$ (ii), one proves Limited Principle of Omniscience, $\forall f : \mathsf{nat} \to \mathsf{bool}.(\forall n : \mathsf{nat}.f(n) = 0) \vee (\exists n : \mathsf{nat}.f(n) = 1)$, stating that any bit-valued function is either everywhere zero or one at some n. However we did not know whether the implication (iv) $\Rightarrow$ (iii) can be proved constructively, and we conjectured that it cannot.

Here we attempt to answer to our conjecture positively. Let us recall the definitions of (iii) and (iv). Given a set $A \subseteq \mathsf{nat}$ of natural numbers, let $N_A \subseteq \mathsf{nat}^*$ be the smallest set of lists over natural numbers that is closed under the following two rules:

(1) any list l over A is in N_A whenever l has a duplicate;

(2) any list l over A is in N_A whenever $a :: l$ is in N_A for every $a : A$.

Then a set A is called *Noetherian* if N_A contains an empty list. This definition is more general than the above (iii), and is equivalent to (iii) when A has decidable equality. A set A is called *streamless* if any stream over A has a duplicate, i.e., $\forall f : \mathsf{nat} \to A.\exists n.\exists m > n.f(n) = f(m)$.

Let H be the halting predicate, that is, $H(i) := \exists k.T(i,i,k)$ with Kleene's T-predicate. We denote binary lists using the notation [...], e.g. [] denotes an empty list, $[1;1;1;1]$ the binary list that contains four 1's, etc. We call a binary list $[b_0;...;b_{n-1}]$ a *partial solution to the halting problem* if $H(i) \iff b_i = 1$ for all $i < n$. Let $\mathcal{P}_H$ be the set of partial solutions to the halting problem. Distinct elements of $\mathcal{P}_H$ have different lengths. We assume binary lists to be encoded as natural numbers in some obvious, recursive way.

The set $\mathcal{P}_H$ is infinite. Indeed we can prove that $N_{\mathcal{P}_H}$ is not Noetherian as follows. Let S be the set of lists over nat containing duplicates. In particular, S contains lists of elements of $\mathcal{P}_H$ encoded as natural numbers. S is clearly closed by (1). To see that S is closed under (2), let ll be a list over $\mathcal{P}_H$ and assume $l :: ll \in S$ for all $l \in \mathcal{P}_H$. It is decidable whether ll has duplicates or not. If so, we have $ll \in S$. If not, we perform case analysis on the shape of ll. Suppose ll is []. Since $[] \in \mathcal{P}_H$, we must have $[[]] \in S$ by our assumption, which contradicts with that the elements of S contain duplicates. So we may assume $ll \neq []$. Let $[b_0;...;b_{n-1}]$ be the longest list in ll. Define $l_i = [b_0;...;b_{n-1};i]$ for $i = 0,1$. By construction, neither $l_0 :: ll$ nor $l_1 :: ll$ has duplicates, hence they are not in S. By assumption, we have that $l' :: ll \in S$ for all $l' \in \mathcal{P}_H$. Hence it follows that $l_i \notin \mathcal{P}_H$ for $i = 0,1$. From the definition of $\mathcal{P}_H$ and that $[b_0;...;b_{n-1}] \in \mathcal{P}_H$, we learn that $\neg(H(n))$ and $\neg\neg(H(n))$, which is absurd. Hence we have $ll \in S$ and S satisfies both the closure conditions defining $N_{\mathcal{P}_H}$. Since S does not contain an empty list, it follows that $N_{\mathcal{P}_H}$ is not Noetherian.

On the other hand, $\mathcal{P}_H$ is streamless for recursive streams. Hence it cannot be disproved that $\mathcal{P}_H$ is streamless. Therefore, the implication (iv) $\Rightarrow$ (iii) is unprovable.

In this talk, I will reconstruct the above (informal) argument in the setting of type theory, by abstracting away the halting predicate.

Acknowledgment Nakata was supported by the Estonian Centre of Excellence in Computer Science, EXCS, financed mainly by the European Regional Development Fund, ERDF, the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12, and the Estonian Science Foundation grant no. 9398.

References

[1] M. Bezem, K. Nakata, and T. Uustalu. On streams that are finitely red. *Logical Methods in Computer Science*, 8(4), 2012.

[2] T. Coquand and A. Spiwack. Constructively finite? In *Scientific contributions in honor of Mirian Andrés Gómez*, pages 217–230. Servicio de Publicaciones, Universidad de La Rioja, Spain, 2010.

Realizability for Peano Arithmetic with Winning Conditions in HON Games

Valentin Blot

Laboratoire de l'Informatique et du Parallélisme
ENS Lyon - Université de Lyon
UMR 5668 CNRS ENS-Lyon UCBL INRIA
46, allée d'Italie
69364 Lyon cedex 07 - FRANCE
valentin.blot@ens-lyon.fr

Abstract

We build a realizability model for Peano arithmetic based on winning conditions for HON games. First we define a notion of winning strategies on arenas equipped with winning conditions. We prove that the interpretation of a classical proof of a formula is a winning strategy on the arena with winning condition corresponding to the formula. Finally we apply this to Peano arithmetic with relativized quantifications and give the example of witness extraction for Π_2^0 formulas.

Realizability is a technique to extract computational content from formal proofs. It has been widely used to analyze intuitionistic systems (for e.g. higher-order arithmetic or set theory), see [12] for a survey. Following Griffin's computational interpretation of Peirce's law [4], Krivine developed in [7, 8, 6] a realizability for second-order classical arithmetic and Zermelo-Fraenkel set theory.

On the other hand, Hyland-Ong game semantics provide precise models of various programming languages such as PCF [5] (a similar model has also been obtained in [10]), also augmented with control operators [9] and higher-order references [1]. In these games, plays are traces of interaction between a program (player P) and an environment (opponent O). A program is interpreted by a strategy for P which represents the interactions it can have with any environment.

In this talk, we devise a notion of realizability for HON general games based on winning conditions on plays. We show that our model is sound for classical Peano arithmetic and allows to perform extraction for Π_2^0 formulas.

HON games with winning conditions on plays have been used in e.g. [3] for intuitionistic propositional logic with fixpoints. Our winning conditions can be seen as a generalization of the ones of [3] in order to handle full first-order classical logic, while [3] only deals with totality. Our witness extraction is based on a version of Friedman's trick inspired from Krivine [8]. Classical logic is handled as in the unbracketed game model of PCF of [9].

We start from the cartesian closed category of single-threaded strategies. This category allows to model references and control operators. We apply the coproduct completion described in [2] to this category and obtain a category of continuations (see [11]). This category enlightens the fact that the usual flat arena of integers in HON games is indeed the negative translation of some coproduct. Our realizability is then obtained by equipping arenas with winning conditions on plays. Similarly to the orthogonality-based framework of Krivine [7], the realizability arrow of our model is not the usual Kleene arrow of intuitionistic realizability (see e.g. [12]).

This work is developed in a paper available at:

http://hal.archives-ouvertes.fr/hal-00793324

References

[1] Samson Abramsky, Kohei Honda, and Guy McCusker. A Fully Abstract Game Semantics for General References. In *LICS*, pages 334–344. IEEE, 1998.

[2] Samson Abramsky and Guy McCusker. Call-by-Value Games. In *CSL*, pages 1–17. Springer, 1997.

[3] Pierre Clairambault. Least and Greatest Fixpoints in Game Semantics. In *FOSSACS*, pages 16–31. Springer, 2009.

[4] Timothy Griffin. A Formulae-as-Types Notion of Control. In *POPL*, pages 47–58. ACM Press, 1990.

[5] J. M. E. Hyland and C.-H. Luke Ong. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[6] Jean-Louis Krivine. Typed lambda-calculus in classical Zermelo-Frænkel set theory. *Arch. Math. Log.*, 40(3):189–205, 2001.

[7] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Theor. Comput. Sci.*, 308(1-3):259–276, 2003.

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

[9] James Laird. Full Abstraction for Functional Languages with Control. In *LICS*, pages 58–67. IEEE, 1997.

[10] Hanno Nickau. Hereditarily sequential functionals. In *LFCS*, pages 253–264. Springer, 1994.

[11] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.

[12] A.S. Troelstra. Chapter VI Realizability. *Studies in Logic and the Foundations of Mathematics*, 137:407–473, 1998.

A Hybrid Linear Logic for Constrained Transition Systems

Kaustuv Chaudhuri[1] and Joëlle Despeyroux[2]

[1] INRIA, Saclay, France
[2] INRIA and I3S - CNRS, Sophia-Antipolis, France

To reason about state transition systems, we need a logic of state. Linear logic [8] is such a logic and has been successfully used to model such diverse systems as process calculi, references and concurrency in programming languages, security protocols, multi-set rewriting, and graph algorithms. Linear logic achieves this versatility by representing propositions as *resources* that are combined using $\otimes$, which can then be transformed using the linear implication ($\multimap$). However, linear implication is timeless: there is no way to correlate two concurrent transitions. If resources have lifetimes and state changes have temporal, probabilistic or stochastic *constraints*, then the logic will allow inferences that may not be realizable in the system being modelled. The need for formal reasoning in such constrained systems has led to the creation of specialized formalisms such as Continuous Stochastic Logic (*CSL*) [2] or Probabilistic *CTL* [9], that pay a considerable encoding overhead for the states and transitions in exchange for the constraint reasoning not provided by linear logic. A prominent alternative to the logical approach is to use a suitably enriched process algebra such as the stochastic and probabilistic π-calculi or the κ-calculus [6]. Processes are animated by means of simulation and then compared with the observations. Process calculi do not however completely fill the need for *formal logical reasoning for constrained transition systems.*

We propose a simple yet general method to add constraint reasoning to linear logic. It is an old idea—*labelled deduction* [13] with *hybrid* connectives [3]—applied to a new domain. Precisely, we parameterize ordinary logical truth on a *constraint domain*: $A@w$ stands for the truth of A under constraint w. Only a basic monoidal structure is assumed about the constraints from a proof-theoretic standpoint. We then use the hybrid connectives of *satisfaction* and *localization* to perform generic symbolic reasoning on the constraints at the propositional level. We call the result *hybrid linear logic* (*HyLL*). *HyLL* has a generic cut-free (but cut admitting) sequent calculus that can be strengthened with a focusing restriction [1] to obtain a normal form for proofs. Any instance of *HyLL* that gives a semantic interpretation to the constraints continues to enjoy these proof-theoretic properties.

Focusing allows us to treat *HyLL* as a *logical framework* for constrained transition systems. Logical frameworks with hybrid connectives have been considered before; hybrid *LF* (*HLF*), for example, is a generic mechanism to add many different kinds of resource-awareness, including linearity, to ordinary *LF* [12]. *HLF* follows the usual *LF* methodology by keeping the logic of the framework minimal: its proof objects are canonical (β-normal η-long) natural deduction terms, where canonicity is known to be brittle because of permutative equivalences [14]. With a focused sequent calculus, we have more direct access to a canonical representation of proofs, so we can enrich the framework with any connectives that obey the focusing discipline. The representational adequacy of an encoding in terms of (partial) focused sequent derivations tends to be more straightforward than in a natural deduction formulation. We illustrate this by encoding the synchronous stochastic π-calculus ($S\pi$) in *HyLL* using rate functions as constraints.

In addition to the novel stochastic component, our encoding of $S\pi$ is a conceptual improvement over other encodings of π-calculi in linear logic. In particular, we perform a full propositional reflection of processes as in [10], but our encoding is first-order and adequate as

in [4]. *HyLL* does not itself prescribe an operational semantics for the encoding of processes; thus, bisimilarity in continuous time Markov chains (*CTMC*) is not the same as logical equivalence in stochastic *HyLL*, unlike in *CSL* [7]. This is not a deficiency; rather, the *combination* of focused *HyLL* proofs and a proof search strategy tailored to a particular encoding is necessary to produce faithful symbolic executions. This exactly mirrors $S\pi$ where it is the simulation rather than the transitions in the process calculus that is shown to be faithful to the *CTMC* semantics [11].

This work has the following main contributions. First is the logic *HyLL* itself and its associated proof-theory, which has a clean judgemental pedigree in the Martin-Löf tradition. Second, we show how to obtain many different instances of *HyLL* for particular constraint domains because we only assume a basic monoidal structure for constraints. Third, we illustrate the use of focused sequent derivations to obtain adequate encodings by giving a novel adequate encoding of $S\pi$. Our encoding is, in fact, *fully adequate* – partial focused proofs are in bijection with traces. The ability to encode $S\pi$ gives an indication of the versatility of *HyLL*. Eventually, we hope to use *HyLL* to encode the stochastic transition systems of formal molecular biology in a sense similar to the κ-calculus [6]. This is the focus of our efforts at the I3S.

The full version of this paper is available as a technical report [5].

References

[1] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

[2] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[3] T. Braüner and V. de Paiva. Intuitionistic hybrid logic. *Journal of Applied Logic*, 4:231–255, 2006.

[4] I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Carnegie Mellon University, 2003. Revised, May 2003.

[5] K. Chaudhuri and J. Despeyroux. A logic for constrained process calculi with applications to molecular biology. Technical report, INRIA, 2010. Available at `http://hal.inria.fr/inria-00402942`.

[6] V. Danos and C. Laneve. Formal molecular biology. *Theor. Comput. Sci.*, 325(1):69–110, 2004.

[7] J. Desharmais and P. Panangaden. Continuous stochastic logic characterizes bisimulation of continuous-time Markov processes. *Journal of Logic and Algebraic Programming*, 56:99–115, 2003.

[8] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[9] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6, 1994.

[10] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *3rd Workshop on Extensions to Logic Programming*, number 660 in Lecture Notes in Computer Science, pages 242–265, Bologna, Italy, 1993. Springer.

[11] A. Phillips and L. Cardelli. A correct abstract machine for the stochastic pi-calculus. *Concurrent Models in Molecular Biology*, Aug. 2004.

[12] J. Reed. Hybridizing a logical framework. In *International Workshop on Hybrid Logic (HyLo)*, Seattle, USA, Aug. 2006.

[13] A. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

[14] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, 2003. Revised, May 2003.

Revisiting the bookkeeping technique in HOAS-based encodings

Alberto Ciaffaglione and Ivan Scagnetto

Università di Udine, Dipartimento di Matematica e Informatica
via delle Scienze, 206 - 33100 Udine, Italia
{alberto.ciaffaglione,ivan.scagnetto}@uniud.it

It is well-known that the advantages of encodings based on Higher-Order Abstract Syntax (HOAS) often thin out as soon as the proof development process starts. In particular, this happens when we want to reason formally about the metatheory of the object language and we must handle, at the proof level, some of the notions delegated to the underlying metalanguage (*e.g.*, bound variables, capture-avoiding substitution). Moreover, we must cope with the lack of support for higher-order induction in many proof assistants. In the literature, there is a lot of work devoted to represent conveniently the binders and to reason formally about object languages with binders, both in first-order and in higher-order settings: layered approaches [GM96], validity predicates [DFH95], nominal calculi [Pit03], axiomatic theories, and so on.

Moreover, HOAS naturally favors a natural deduction-style representation of logical systems, as opposed to common sequent-style presentations on paper. Indeed, this allows for shallow encodings of proof contexts, which are no longer rendered via explicit lists of, *e.g.*, type assumptions (for instance, let us think about formalizations of type systems); instead, the assumptions contained in a classic proof context are accommodated by means of an auxiliary "bookkeeping" judgment [Mic97], which simply records the existence of such assumptions. As a result, the final encoding is, without the usual list machinery, rather terse and clear to read. However, as in the case of the syntax representation, many issues rise up when it is the moment to reason formally about the metatheory of the object language; for instance, we must resort to alternative proof techniques in place of the usual inductions over the length of the contexts.

In this paper, we present an ongoing formal development addressing both the problem of providing an elegant and concise encoding of object languages with binders, and the issue of allowing the user to reason formally about their metatheory in a rather straightforward way. The object language that we adopt as the benchmark is the type language of pure System $F_{<:}$ (already used as a test-bed for the famous POPLMark Challenge [ABF$^+$05, ABF$^+$]).

In fact, we carry out a weak-HOAS encoding of System $F_{<:}$ in the `Coq` proof assistant, and we exploit the Theory of Contexts (ToC: an axiomatization of some simple properties about variables, see [HMS01, BHM$^+$06]) to formally prove statements involving binders. In particular, we introduce an implicit representation of the typing context based upon the bookkeeping technique, as opposed to the explicit list-based one presented in [CS12].

To reconcile weak-HOAS with `Coq`'s (co)inductive features, we use a separate non-inductive type `Var` for variables; the novelty of our rendering of the context Γ in judgments of the form $\Gamma \vdash M : \sigma$ consists in the introduction of the following parametric[1] predicate `envTp`:

```
Parameter envTp: Var -> Tp -> Prop.
```

Thus, given a System $F_{<:}$ environment $\Gamma = \{X_1{<:}T_1, \ldots, X_n{<:}T_n\}$, its formal representation in `Coq` amounts to the following set of assumptions:

```
Parameter h1: (envTp X1 T1). ... Parameter hn: (envTp Xn Tn).
```

[1] In `Coq`, the `Parameter` declaration allows the user to introduce a *global* variable.

where, of course, we denote by `Ti` the `Coq` encoding of T_i. So doing, the representation of a typing environment is not confined to a list, but it is "global", in the sense that every variable declaration is always accessible. This implies that we can encode the subtyping relation as an inductive predicate `subTp: Tp -> Tp -> Prop`, without carrying around the environment.

Obviously, there are some drawbacks to deal with. First, being `envTp` an *open* predicate, we must enforce by some `Axiom`s the usual good formation conditions about contexts. Secondly, a more subtle issue must be handled; for instance, let us consider the informal narrowing statement:

$$\Gamma, X{<:}Q, \Delta \vdash M{<:}N \wedge \Gamma \vdash P{<:}Q \Rightarrow \Gamma, X{<:}P, \Delta \vdash M{<:}N.$$

It is apparent that the variable typings $X{<:}Q$ and $X{<:}P$ belong to two distinct environments, which are involved in distinct derivations. Instead, a naive formalization in `Coq` would introduce two assumptions, `(envTp X Q)` and `(envTp X P)`, in the same *global* environment, yielding two different typing assumptions for the same variable (*i.e.*, a non well-formed environment).

To avoid these inconsistencies, we use distinct (non-clashing) variables (`notin_ho` encodes the "non-occurrence" predicate of a variable in an abstracted term, *i.e.*, a term with a "hole"):

```
Theorem narrowing: forall X:Var, forall M N: Var->Tp, forall Q P:Tp,
  (notin_ho X M) -> (notin_ho X N) -> (envTp X Q) -> (subTp (M X) (N X)) ->
  (subTp P Q) -> forall Y:Var, ~X=Y -> (notin_ho Y M) -> (notin_ho Y N) ->
  (envTp Y P) -> (subTp (M Y) (N Y)).
```

According to our experiments with these kinds of encodings, the implicit representation of contexts Γ works well with the ToC, because we can derive specialized induction principles which are more powerful (from a practical point of view) than `Coq`'s built-in `induction` tactic.

In fact, the trick for System $F_{<:}$ is to associate a "measure" to subtyping derivations (*e.g.*, the number of rules used), a technicality which allows to recover a suitable induction principle for dealing with metaproperties involving variable renamings (rendered as functional applications), like in the above narrowing theorem.

References

[ABF+] B. E. Aydemir et al. The POPLmark Challenge. `http://www.seas.upenn.edu/~plclub/poplmark/`.

[ABF+05] B. E. Aydemir et al. Mechanized Metatheory for the Masses: The POPLmark Challenge. *In Proc. of TPHOLs*, LNCS 3603, pp. 50–65, 2005.

[BHM+06] A. Bucalo, F. Honsell, M. Miculan, I. Scagnetto, and M. Hofmann. Consistency of the Theory of Contexts. *J. of Funct. Program.*, 16(3):327–372, 2006.

[CS12] A. Ciaffaglione and I. Scagnetto. A weak HOAS approach to the POPLmark Challenge. *In Proc. of LSFA*, 2012. To appear.

[DFH95] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-Order Abstract Syntax in Coq. *In Proc. of TLCA*, LNCS 902, pp. 124–138, 1995.

[GM96] A. D. Gordon and T. Melham. Five Axioms of alpha-conversion. *In Proc. of TPHOLs*, LNCS 1125, pp. 173–190, 1996.

[HMS01] F. Honsell, M. Miculan, and I. Scagnetto. The Theory of Contexts for First-Order and Higher-Order Abstract Syntax. *Electr. Notes Theor. Comput. Sci.*, 62:116–135, 2001.

[Mic97] M. Miculan. Encoding Logical Theories of Programs. *PhD thesis*, Dipartimento di Informatica, Università di Pisa (Italy), 1997.

[Pit03] A. M. Pitts. Nominal Logic, a First-Order Theory of Names and Binding. *Information and Computation*, 186:165–193, 2003.

Lightweight proof by reflection using a posteriori simulation of effectful computation*

Guillaume Claret[1], Lourdes del Carmen González Huesca[1], Yann Régis-Gianas[1] and Beta Ziliani[2]

[1] PPS, team πr^2 (University Paris Diderot, CNRS, and INRIA),
{guillaume.claret,lgonzale,yann.regis-gianas}@pps.univ-paris-diderot.fr
[2] Max Planck Institute for Software Systems (MPI-SWS),
beta@mpi-sws.org

The Coq proof assistant, defined as a *formal proof management system*, is a well-established system for developing and checking proofs. It is based on a formal language –the Calculus of (co)-Inductive Constructions– rooted in Type Theory. It generates proof-terms interactively, which are certified in the end of their development by a small typechecker called the kernel. The proof technique *proof by computation* replaces parts of proofs by internalizing them as computations, a benefit from Type Theory in which types may embed computation. This allows to replace potentially large hand-written proof-terms by small proof-terms whose verification requires computation at type level. Many developments in Coq use a successful instance of this technique, *proof by reflection*, based on decision procedures on reified formulas.

There are several advantages and weaknesses of proof by reflection, especially in comparison with the traditional LCF proof style. Indeed, the development of proofs using this technique has a price; for a decision procedure, say D, two things must be taken into account: first, writing D, for example in Coq, restricts the programmer to only use total functions; second, the proof of soundness of D could require extra work that may be of greater difficulty than the proof of the original goal itself. As a result, the implementation of decision procedures is sometimes over-simplified to abbreviate the proof of soundness, which may lead to inefficiencies. This is regrettable because proof search is intrinsically a computationally expansive process.

Some variants of the initial technique were invented to facilitate proving by reflection. The main differences between them are: (i) the language in which the decision procedure D is programmed; (ii) the way the soundness proof is obtained; (iii) the shape of the final proof-term. In the original style of proof by reflection, the programming language does not have native imperative features like side-effects and usually is a total programming language. The final proof-term is a proof of equality, which is small and its type-checking is just a convertibility check. In a certifying style, the decision procedure is written in a general purpose language, typically an effectful language, and is mostly used as an untrusted but sophisticated oracle by the theorem prover. The proof of soundness is done by a certificate-checker, which should prove to be sound. The final proof-term has to embed the certificate, therefore it can not be small as in the original style.

The aim of our work consists in a novel lightweight style of proof by reflection. Our idea is to use *an (untrusted) compiled version of a monadic decision procedure written in Type Theory within an effectful language, as an efficient oracle for itself.* The decision procedures are written in a language based on Type Theory, which is extended with monads as commonly found in Haskell programs. In this way, programmers have a full set of effects at their hands (references, exceptions, non-termination), together with dependent types to enforce (partial) correctness. The evaluation of decision procedures, in our *monadic style* inside Coq, is designed

*Accepted paper in ITP 2013.

to be executed with the help of what we call a *prophecy*. The decision procedure is compiled into an impure programming language (OCaml) with an efficient computational model which performs all the effectful computations. The compilation maps the computations of the monad in Coq, to effectful terms of OCaml; it also instruments the code to compute a small piece of information to efficiently simulate a converging reduction of the compiled code in Type Theory, using the initial monadic decision procedure. Finally, a relation of *a posteriori* simulation stands between the compiled term $\mathcal{C}(t)$ and the initial monadic term t.

To formalise this idea in a general way, we have studied a concept of *a posteriori* simulation of effectful computations in Type Theory. Roughly speaking, given a computation C of type $\mathsf{M}\,\mathsf{A}$, we determine on which conditions there exists a piece of information p such that the evaluation of C using p can witness an inhabitant of type A. Intuitively, if a compiled computation $\mathcal{C}(t)$ converges to a value v, then the same evaluation can be simulated *a posteriori* in Coq using some prophecy p. This prophecy *completes* the computation to get a reduced one "$\Downarrow_p t$" convertible to "$\mathsf{unit}\,t'$", for some term t' of type T. The formalization of the principle of *a posteriori* simulation focuses on simply typed λ-calculus. On the one hand, we define λ, a purely functional and strongly normalizing programming language parameterized by a monad M, which is abstractly specified by a set of requirements. On the other hand, we define $\lambda_{v,\bot}$, an impure functional and non-terminating programming language. The language λ includes the usual monadic combinators for effects in the spirit of [5] and two unusual expressions: the constant $\Downarrow$ and the type constructor P for prophecies. The role of $\Downarrow$ is to perform *a posteriori* simulation, therefore we read $\Downarrow_p t$ of type $\mathsf{M}\,\mathsf{T}$ as "the reduced computation of t using the prophecy p". This reduction remains as a computation, so $\Downarrow$ has type $\mathsf{P} \to \mathsf{M}\,\mathsf{T} \to \mathsf{M}\,\mathsf{T}$. The type constructor M, used as a parameter for λ, is a *simulable monad* if the simulation of effectful constants $\boldsymbol{c}$ converges thanks to the prophecies obtained during execution of the compilation of $\boldsymbol{c}$. The main theorem relates small-step semantics of λ and big-step semantics of $\lambda_{v,\bot}$: *let* $\cdot \vdash t : MT$ *be a computation which compilation converges to a value and produces a prophecy* p'*, then there exists a term* t' *such that* $\Downarrow_{p'} t = \star t'$.

There is a prototype plugin[1] for Coq to develop proofs using the method we described. The monad includes the effectful operations of partiality, non-termination, state and printing. Also the non-determinism can be programmed on top of the monad. Its type definition is parameterized by a signature to type the memory operations: $\mathsf{M}\ \Sigma\ \alpha = \mathsf{State.t}\ \Sigma \to (\alpha + \mathsf{string}) \times \mathsf{State.t}\ \Sigma$. We keep the power of the dependently-typed system of Coq despite the fact that we are working in a monad. Using this proposal of *proof by reflection* we also get a great performance, gained for type-checking. This is an ongoing work, our long-term goal is to build upon this system to use Coq as a typed tactic language for Coq.

References

[1] Guillaume Claret, Lourdes del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. Technical report, 2013. https://gforge.inria.fr/frs/download.php/32051/coqbottom-techreport.pdf.

[2] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[3] Dimitri Hendriks. Proof reflection in coq. *Journal of Automated Reasoning*, 29(3-4):277–307, 2002.

[4] Martijn Oostdijk and Herman Geuvers. Proof by computation in the coq system. In *Theoretical Computer Science*, pages 293–314. Elsevier, 2000.

[5] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[1] http://coqbottom.gforge.inria.fr

Computable Refinements by Quotients and Parametricity*

Cyril Cohen[1], Maxime Dénès[2] and Anders Mörtberg[1]

[1] Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
{cyrilc,mortberg}@chalmers.se
[2] INRIA Sophia Antipolis – Méditerranée
Maxime.Denes@inria.fr

It is commonly conceived that computationally well-behaved programs and data structures are usually more difficult to study formally than naive ones. Rich formalisms like the Calculus of Inductive Constructions, on which the Coq [2] proof assistant relies, allow several different representations of the same object so that users can choose the one suiting their needs.

Even simple objects like natural numbers may have both a unary representation which features a very straightforward induction scheme and a binary one which is exponentially more compact, but usually entails more involved proofs. Their respective incarnations in the standard library of Coq are the two inductive types `nat` and `N` along with two isomorphisms `N.of_nat : nat -> N` and `N.to_nat : N -> nat`. Operators and proofs used to be duplicated over these two types, whereas recent versions of the library make use of functors and type classes to factor the code.

However, this traditional approach to abstraction, by defining an interface specifying the signature of operators and axiomatizing their properties, proving correctness from the axioms, and then instantiating to particular implementations, has at least two drawbacks in our context. First, it is not always obvious how to define the correct interface, and not even clear that a suitable one always exists. Second, having abstract axioms goes against the type-theoretic view of objects with computational content which means in practice that proof techniques like small scale reflection, as advocated by the SSReflect extension [4], are not applicable. Instead, the approach we propose involves proving the correctness of operators on data structures designed for proofs – as opposed to an abstract signature – and then transporting them to more efficient implementations. We distinguish two notions: *program refinements* and *data refinements*.

The first of these consists in transforming a program into a more efficient one computing the same thing, preserving the involved types. For example, standard matrix multiplication can be refined to a more efficient implementation like Strassen's fast matrix product. The correctness of this kind of refinements is often straightforward to state in Coq as, in many cases, it suffices to prove that the two algorithms are extensionally equal.

However, we consider program refinement as an input to our framework and we do not explain here how to prove them formally. Instead, we focus on data refinements, by which we mean linking two different datatypes representing the same mathematical objects, and lifting the associated operators from one to the other.

We rely on a previous work [3] which defines a framework refining *algebraic structures* in a similar way, while allowing a step-by-step approach to prove correctness of algorithms. The present work[1] improves generality by extending to previously unsupported data types (like

*The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

[1] Documentation and development is available at `http://www.maximedenes.fr/coqeal/`

the rational numbers), flexibility and modularity by refining each *operator* in isolation, and automation.

In practice our framework decomposes each library into three parts. The first part consists in defining the computation-oriented datastructures together with the (possibly efficient) algorithms operating on it. Both may be based on abstract types and operators for the underlying structures. For example, given an abstract type `A` equipped with a binary operator `mul : A -> A -> A` representing multiplication, rational numbers may be implemented by the type `A * A` of pairs of elements in `A`. It is then possible to define multiplication for this representation. In order to make programming easier we use operational typeclasses [7] to parametrize by operations such as `mul`.

In the second part, the correctness of the computation-oriented type and its operations is proved by linking them to their proof-oriented counterparts. For rational numbers this could for example mean that the representation as a pair would be linked to the type `rat` of rational numbers of SSReflect, which are pairs of coprime integers where the second component is nonzero. In this case `A * A` and `rat` would not be isomorphic, but the latter would be a partial quotient of the former [1].

The correctness of the implemented algorithms is proved by instantiating by proof-oriented types and is proved with respect to its proof-oriented counterpart. For rational numbers, this would lead us to change the abstract type `A` to a proof-oriented representation of integers and prove that the definition of multiplication computes the same thing as the one for `rat` (modulo normalization). These correctness proofs are stored in a database (using typeclass instances [6]) so that they can be instrumented for automated translation.

The third step is to substitute the proof-oriented parameters with computation-oriented ones. Applying it on rational numbers implies changing the proof-oriented representation of integers into a computation-oriented one, which refines the former. This step can be done by taking advantage of parametricity. However since parametricity is not internal in Coq [5], we rely on meta-programming.

This work is intended to be used as a new basis for CoqEAL – The Coq Effective Algebra Library. This is a library that we are currently developing, containing many formally verified program refinements such as Strassen's fast matrix product, Karatsuba's fast polynomial product and the Sasaki-Murao algorithm to compute the characteristic polynomial of a matrix.

References

[1] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory.* Phd thesis, Ecole Polytechnique X, November 2012.

[2] Coq development team. The Coq Proof Assistant Reference Manual, version 8.4, 2012.

[3] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A Refinement Based Approach to Computational Algebra in Coq. In *ITP*, volume 7406 of *LNCS*, pages 83–98, 2012.

[4] Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009. `http://hal.inria.fr/inria-00258384`.

[5] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In *CSL - 26th International Workshop/21st Annual Conference of the EACSL - 2012*, volume 16, pages 381–395, 2012.

[6] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 278–293, 2008.

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

Formalizing Subsumption Relations in Ontologies using Type Classes in Coq

Richard Dapoigny and Patrick Barlatier

LISTIC/Polytech'Annecy-Chambéry, University of Savoie,
Po. Box 80439, 74944 Annecy-le-vieux cedex, France
Tel.: +33-45-096529 Fax: +33-45-096559

Abstract

While subsumption is a crucial reasoning tool in ontologies as well as in Conceptual Modeling, its formal definition is not precise and its implementation in usual first-order-logic-based theories suffers from a lack of expressiveness and leads to many ambiguities. We show in this talk that the Coq language (based on the Calculus of Inductive Constructions with type classes) extended with an ontological layer is able to support both an expressive and clear semantics for the subsumption relations, namely is_a and part-whole.

Summary

The most widely discussed relations are formal relations and more precisely, class subsumption (is_a relation) and partonomic inclusion (part-whole relation). Both is_a and part-whole are ubiquitous in foundational ontologies and many works have investigated their representation. However, due to imprecise definitions of their real-world semantics, the use of these relations and particularly of $part-whole$ relation is often problematic as a way of communicating meaning in an application domain. For example, although Description Logics (DL) is being constantly refined and extended, it can hardly properly define the mereological parthood relation[1] which is essential e.g., in many biological ontologies and even $\mathcal{SROIQ}$ can only partially capture mereology. Whilst proper parthood can be captured as a symmetric, transitive, and asymmetric relation, $\mathcal{SROIQ}$ cannot express anti-symmetry, and parthood cannot adequately be modeled. Therefore, in this paper we address the problem of defining ontological relations in a more rigorous and unambiguous way. For that purpose we make use of a unified language [2], i.e., K-DTT (Knowledge-based Dependent Type Theory) able to construct very expressive hierarchies. In particular we will demonstrate that properties of relations propagate through the inheritance hierarchy, resulting in a significant simplification of the reasoning process. We restrict the study to the more foundational relations, i.e., is_a and part-whole relations since it is the basis of most existing works about foundational ontologies.

The Calculus of Constructions with inductive types and universes has given rise to the Coq language, a theorem prover for developing mathematical specifications and proofs. We use powerful primitives such as Types Classes (TCs) for describing data structures. TCs in Coq are similar to type classes in Haskell, however Coq allows us to do better by specifying the rules inside TCs. TCs allow parametric arguments, inheritance and multiple fields, then, any meta-model can be described with TCs. Since TCs can be predicates, one has to prove the type class with instances. It follows that instances are nothing else than models proving the specifications of the TC w.r.t. semantics. We restrict here the scope to the formalization of relations, described with TCs. We follow the idea of using an ontology for controlling the

[1] It refers here to the transitive part-of relation as specified in mereology

semantics of formal relations and will show how a well-founded conceptual model for these relations can be designed (e.g., with specifications for relation types).

Whereas usual conceptual modeling languages separate the *is_a* and the part-whole relations, we claim that a general subsumption relation called *mereosis*, is able to encompass both relations. The differences between these relations relies on the notion of attribute[2] persistence [1]. Suppose that we have a mathematical space S in which parts are ordered by the inclusion relation $\subseteq$. Given two parts U and V, they are assigned the respective collections $A(U)$ and $A(V)$ of attributes owned by each of them. The possession of primitive attributes is said persistent if $U \subseteq V \rightarrow A(V) \subseteq A(U)$. In such a way, *is_a* relations have the persistence property while part-whole relations do not possess this property. Such properties are expressed with specifications, i.e., TCs involving a structure and axioms about this structure. Using parametric universes, coercive subtyping and inheritance between TCs, a hierarchy can be arranged between specifications. Specifying *is_a* relations with persistence also complies with Formal Concept Analysis (FCA) which is a theory of data analysis identifying conceptual structures among data sets. The subconcept/superconcept relation gives the same constraints provided that (i) the set of attribute is the set of properties of a universal and (ii) the set of objects corresponds to the set of proof objects that fall under the universal. It follows that the basic structure describing foundational *is_a* relations is a lattice and supports multiple inheritance.

The strong property of non-persistence must be refined in order to cope with the multiple constraints which inhere in the formalization of part-whole relations. A major distinction underlying a clear part-whole classification relies on the transitivity property of the part-whole relation while a second distinction should be based on the types of their relata. Dedicated specifications express that the part-whole relation can be either transitive, intransitive or non-transitive, i.e., neither transitive nor intransitive. Furthermore, different types of the part-whole relations are captured according to the type of their relata. Then, specifications of part-whole properties arranged into TCs will form a hierarchy using multiple inheritance. Wider inheritance hierarchies can be investigated for describing expressive knowledge hierarchies in which type checking allow for proving well-formed ontologies. It is worth noticing that the number of specifications is quite limited and then, that the computing properties such as decidability can be preserved.

Usual theories about part-whole relations do not consider categories of the entity types involved in a part-whole relation and any *is_a* relation between these relations presuppose that their arguments are identical. The K-DTT language is able to control both the types of each argument if required and inheritance between the type classes representing distinct relation types. This property extends the expected expressiveness of the theory beyond the usual power of ontology languages since it involves not only the predicates of relation types (i.e., specifications) but also all arguments for these types.

References

[1] J. L. Bell, Whole and Part in Mathematics, Axiomathes, 14 pp 285–294, Kluwer Academic Publishers, (2004).

[2] P. Barlatier, R. Dapoigny, A Type-Theoretical Approach for Ontologies: the Case of Roles, Applied Ontology, 7(3), IOS Press (2012) pp 311–356.

[2] Also called here properties.

Mechanized semantics for an Algol-like language

Daniel Fridlender, Miguel Pagano and Leonardo Rodríguez

{fridlend,pagano,lrodrig2}@famaf.unc.edu.ar
Universidad Nacional de Córdoba,
Córdoba, Argentina.

In the last decade there has been several important achievements in the field of compiler certification, of which the project CompCert [9, 10] is the most noteworthy because it deals with a realistic compiler for C. In spite of these advances it is an active research topic open for new techniques and experimentation: for example, can one use denotational semantic models to structure the proof of correctness?

Reynolds [15] defined an intermediate language generator for an Algol-like language using functorial categories, and suggested it as an approach prone to compiler certification. Our long-term goal is to develop a certified compiler for a full-fledged Algol-like language with a non-trivial type system along the lines of Reynolds' proposal. In this talk we report advances in that direction: we have proved in Coq [1] the correctness of a compiler for a small language featuring the main characteristics of the Algol tradition: a simple imperative language, a procedure mechanism based on a typed call-by-name lambda calculus, and a stack-discipline for the allocation of variables in the store.

Leroy [8] defined an abstract machine for a call-by-value lambda calculus, and used coinductive big-step semantics to describe the behaviour of divergent programs. He also used Coq to prove the correctness of the compiler and some additional semantic properties like evaluation determinism and progress. A similar approach has been used by Leroy [11] and Bertot [3] for imperative languages.

We have followed the same strategy, but with significant differences due to the characteristics of Algol-like languages: (i) the call mechanism is call-by-name rather than call-by-value; (ii) it combines imperative and applicative features; (iii) the store allocation mechanism follows a stack discipline: local variables are deallocated at the end of their block.

Another difference with respect to [8, 11, 3] is the use of information provided by the type system at compilation time, inserting appropriate instructions on the generated code. In this way we were able to simplify our previous attempts of designing our compiler and abstract machine where such information was available only at run-time.

We are still far from our goal of having a certified compiler for Forsythe [16]: our next milestone in that direction is the inclusion of intersection types. At the same time we plan to have a better understanding of Reynolds' proposal of using denotational semantics to prove the correctness of the compiler; the first step in this path is to identify a class of denotational models for the language, one of which would be the intermediate language which we are targeting.

Related Work. The semantic models for Algol-like languages have been studied by several authors including Reynolds [17], Oles [12, 13] and more recently by Reddy [14]. Reynolds [15] presented a compiler for a small Algol-like language, defined as a special case of his functorial semantic model. Harrison [6, 7] used partial evaluation to generate code, having the compiler of [15] as a starting point. Chlipala [4, 5], and Bertot [2] have worked on computer-assisted verification of compilers for several programming languages, including typed lambda calculus and impure functional languages.

References

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

[2] Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, September 1998.

[3] Yves Bertot. Theorem proving support in programming language semantics. *CoRR*, abs/0707.0926, 2007.

[4] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, 43(9):143–156, September 2008.

[5] Adam Chlipala. A verified compiler for an impure functional language. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 93–106. ACM, 2010.

[6] William L. Harrison and Samuel N. Kamin. Compilation as partial evaluation of functor category semantics. Technical report, University of Illinois, Urbana-Champaign, 1997.

[7] William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the 1998 International Conference on Computer Languages*, ICCL '98, pages 122–, Washington, DC, USA, 1998. IEEE Computer Society.

[8] Xavier Leroy. Coinductive big-step operational semantics. In *European Symposium on Programming (ESOP 2006)*, pages 54–68. Springer, 2006.

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

[10] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[11] Xavier Leroy. Mechanized semantics - with applications to program proof and compiler verification. In *Logics and Languages for Reliability and Security*, pages 195–224. 2010.

[12] Frank J. Oles. *A category-theoretic approach to the semantics of programming languages*. PhD thesis, Syracuse University, Syracuse, NY, USA, 1982. AAI8301650.

[13] Frank J. Oles. Type algebras, functor categories and block structure. In Maurice Nivat and John C Reynolds, editors, *Algebraic methods in semantics*, pages 543–573. Cambridge University Press, New York, NY, USA, 1986.

[14] Uday S. Reddy and Brian P. Dunphy. An automata-theoretic model of idealized Algol. In *Proceedings of the 39th international colloquium conference on Automata, Languages, and Programming - Volume Part II*, ICALP'12, pages 337–350, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] John C. Reynolds. Using functor categories to generate intermediate code. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 25–36, New York, NY, USA, 1995. ACM.

[16] John C. Reynolds. Design of the programming language Forsythe. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like Languages, Volume 1*, pages 173–233. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[17] John C. Reynolds. The essence of Algol. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like Languages, Volume 1*, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

Nested Typing and Communication in the λ-calculus

Nicolas Guenot

IT University, Copenhagen, ngue@itu.dk

The Curry-Howard correspondence allows to ensure through a type system that a given λ-term is safe to execute, in the sense that it will terminate, and it has been extended to settings using the cut rule to type explicit substitutions [1], which allow to decompose β-reduction and are closer to the actual implementation of functional programming languages, related to abstract machines. It has also been extended to handle erasure and duplication of pieces of code explicitly [4], as should be done in a realistic implementation, and generalised to lists of arguments by using the sequent calculus as a basis for the correspondence [3].

Since reduction in the λ-calculus can be performed following different strategies, for example by starting with functions bodies as in call-by-name or with arguments as in call-by-value, we know that it is possible to reduce separate parts of a λ-term in parallel. In the reduction:

$$(\lambda x.t\ v)\ u\ \longrightarrow_{\mathsf{B}}\ (t\ v)[x \leftarrow u]\ \longrightarrow^*\ t[x \leftarrow u_1]\ v_1$$

where x does not appear in v, it is clear that the reductions $u \longrightarrow^* u_1$ and $v \longrightarrow^* v_1$, as well as the pushing of the substitution inside the application, can be performed in parallel, since confluence ensures that the result is the same in any possible interleaving. However, this parallelism appears in the fact that cuts can be picked independently to be reduced, in different branches, but it cannot be exploited to introduce *concurrency*: it would require to have synchronisation points where reduction in one branch could interfere with reduction in another branch, which cannot happen.

The purpose of the work presented here is to use a particular feature of proof systems following the *deep inference* methodology [2] to introduce a notion of *communication* in a typed λ-calculus, in a way that allows computation to be distributed among several units executing not only in parallel, but also concurrently in the sense that they must communicate resources to one another to perform computation. The logical system used as a basis in this correspondence is a new system we call JD for the implication-only fragment of intuitionistic logic, in the calculus of structures. It is designed as an adaptation of the natural deduction system NJ to the setting of deep inference, and uses two connectives for implication — the object-level implication $\to$ and the meta-level implication $\supset$ — as shown below:

$$A, B\ ::=\ a\ \mid\ \mathtt{t}\ \mid\ A \to B\ \mid\ A \supset B \qquad \begin{cases} \mathtt{t} \supset A & \equiv & A \\ A \supset (B \supset C) & \equiv & B \supset (A \supset C) \end{cases} \tag{1}$$

where the equations on the right generate a congruence $\equiv$ on formulas, so that structures in this system can be defined as equivalence classes of formulas, as usual in the calculus of structures. The specific inference rules used are the following:

$$\mathsf{x}\,\frac{}{A \supset A}\qquad \mathsf{i}\,\frac{A \supset B}{A \to B}\qquad \mathsf{w}\,\frac{B}{A \supset B}\qquad \mathsf{s}\,\frac{((A \supset B) \supset C) \supset D}{A \supset (B \supset C) \supset D}$$
$$\mathsf{e}\,\frac{(A \supset A) \to B}{B}\qquad \mathsf{c}\,\frac{A \supset A \supset B}{A \supset B}\qquad \mathsf{si}\,\frac{((A \supset B) \supset C) \to D}{A \supset ((B \supset C) \to D)} \tag{2}$$

where the rules x, i, w and c are essentially the same rules as in NJ, but where the elimination rule e is decomposed, so that no hypothesis is given to prove A, as hypotheses must be pushed one by one, in a lazy way, to the left of A using the switch rules s and si.

As in natural deduction, normalisation can be proved for JD using a rewriting procedure based on permutation of rule instances, using the cut, defined as the composition of the i and e rules. This can be used as a basis for an adaptation of the Curry-Howard correspondence to this setting, and using for example a variant of JD where the switch rules are built inside the elimination rule, we obtain a type system for standard λ-calculi with explicit substitutions, such as λs [5].

When using JD as the basis for a type system, we have to provide a computational interpretation of the switch rules, in the sense that they should be used to type some operators in an extended λ-calculus. The crucial observation is that the standard elimination rule is translated to this setting using several switches, as follows:

$$\rightarrow_{\mathsf{e}} \frac{C_1, \cdots, C_n \vdash A \qquad \Gamma \vdash A \to B}{\Gamma, C_1, \cdots, C_n \vdash B} \qquad \longrightarrow \qquad \mathsf{si} \frac{\mathsf{si} \frac{\cdots}{\Gamma \supset (((C_1 \supset \cdots \supset C_n \supset A) \supset A) \to B)} \cdots}{\mathsf{si} \frac{\Gamma \supset C_1 \supset \cdots \supset (((C_n \supset A) \supset A) \to B)}{\mathsf{e} \frac{\Gamma \supset C_1 \supset \cdots \supset C_n \supset ((A \supset A) \to B)}{\Gamma \supset C_1 \supset \cdots \supset C_n \supset B}}}$$

but in some given proof in JD, all these switches are not necessarily located immediately above the elimination instance. This means that one switch instance corresponds to an operator used to provide one resource — a typing assumption — to a subterm typed deeper in the structure. A way of representing this is to type two operators for *input* and *output* applied on the same *channel*:

$$\mathsf{si} \frac{t : ((\Delta, y : A \supset u : B) \supset C) \to D}{x : A \supset \overline{\alpha} x.t : ((\Delta \supset \alpha y.u : B) \supset C) \to D} \tag{3}$$

where $\overline{\alpha} x.t$ is a term that sends on α an explicit substitution on x, seen as a resource, and $\alpha y.u$ is receiving a resource on α and binding it to y, much like in the (higher-order) π-calculus [6]. The normalisation procedure for JD yields a set of rewrite rules where substitutions are always pushed on the left in applications, and including communication rules:

$$\begin{array}{rcl} (t\ v)[x \leftarrow u] & \longrightarrow & t[x \leftarrow u]\ v \\ (\overline{\alpha} x.t)[x \leftarrow u]\ (\alpha y.v) & \longrightarrow & (\overline{\alpha_i}\phi.t)\ (\alpha_i \phi.v[y \leftarrow u]) \end{array} \tag{4}$$

where ϕ is the list of free variables of u and the channels α_i are fresh. Another communication rule is responsible for passing a resource to a term inside another substitution. The resulting λ-calculus, called λc, provides a view of explicit substitutions as resources, and of subterms in applications as concurrent processes communicating through channels. Unfortunately, the operational behaviour of the untyped calculus is rather complicated, in particular because of *"backwards references"* and *deadlocks*, but typing ensures that a term can be reduced — and it might also ensure confluence.

References

[1] M. Abadi, L. Cardelli, P-L. Curien, and J-J. Lévy. Explicit substitutions. In *POPL'90*, pages 31–46, 1990.

[2] A. Guglielmi. A system of interaction and structure. *ACM TOCL*, 8(1):1–64, 2007.

[3] H. Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *CSL'94*, volume 933 of *LNCS*, pages 61–75, 1994.

[4] D. Kesner and S. Lengrand. Resource operators for the λ-calculus. *Information and Computation*, 205(4):419–473, 2007.

[5] D. Kesner and F. Renaud. A prismoid framework for languages with resources. *Theoretical Computer Science*, 412(37):4867–4892, 2011.

[6] D. Sangiorgi. From π-calculus to higher-order π-calculus - and back. In M-C. Gaudel and J-P. Jouannaud, editors, *TAPSOFT'93*, volume 668 of *LNCS*, pages 151–166, 1993.

Some reflections about equality in type theory (tentative material for further research)

Hugo Herbelin

INRIA - PPS - University Paris Diderot

Type theory comes with various notions of equality: judgemental vs propositional, intensional/definitional vs extensional, with or without functional extensionality, with or without uniqueness of equality proof, with or without a univalent notion of equality of types [11][1]

Consider for instance the case of Intensional Type Theory (ITT) and Extensional Type Theory (ETT). In ITT, intensional equality is defined in a purely judgemental way and is used for conversion. Built on top of an initial set of definitions, it satisfies functional extensionality (ξ-rule of λ-calculus) and uniqueness of (de facto invisible) equality proofs. Reified into the identity type connective, it extends into a propositional equality and gets closed under extensional (inductive and hypothetical) reasoning. In ETT, propositional equality is reflected back into judgemental equality: they start to share their properties ending in both being extensional (i.e. closed under inductive and hypothetical reasoning) and both satisfying functional extensionality and uniqueness of proofs, in agreement with an old result by Hofmann [6].

None of these is satisfactory. In practice, ITT offers a too weak judgemental equality compared to what users of proof assistant, or even informal type theoretists would expect. For instance, $0+x = x+0$, though obvious, is not judgemental. On the other side, $0+x = x+0$ is part of judgemental equality in ETT but there, judgemental equality is undecidable which potentially breaks normalisation and mechanisation of reduction in unsatisfiable typing contexts.

The works of Blanqui, Jouannaud, Okada on the Calculus of Algebraic Constructions [4], later extended by works with Strub on Coq Modulo Theory [5, 10], suggest that judgemental equality does not have to be either intensional or extensional, but instead "technological" in the sense that the principal criterion that could justify it is its ability to be mechanically decided.

This suggests to explore how to fully get rid of the judgemental equality artefact by rephrasing judgemental equality as a subset of propositional equality in ways similar to what Oury [8] did for interpreting the extensional Calculus of Constructions into the intensional one. This requires axiomatising the definitional rules (β, etc.), and adding further axioms for simulating the expected "invisibility" of these extra axioms. Then, once the judgemental equality artefact and its associated conversion rule are ruled out, it is possible to reintroduce a conversion rule with transparent trace of equality along the sole rationale of decidability of the reconstruction of an equality evidence, as done in Coq Modulo Theory.

Other questions arise regarding the computational content of functional extensionality and univalence. Based on Altenkirch, McBride and Swierstra's work on Observational Type Theory [2, 3] and on Licata and Harper's work on the computational content of univalence [7], we propose to investigate how to give a confluent type-safe computational content to functional extensionality and univalence, ensuring canonicity for data-types under the extra assumption of normalisation.

Finally, we address the question of providing a theory with both a univalent and a strict extensional equality coexisting, this latter point being initially motivated by a formalisation of simplicial types in homotopy-type-theoretic Coq which would not be possible for types of

[1] One might also contrast typed judgemental equality and untyped judgemental equality, but based on results such as [1, 9], we will make the assumption that this difference is a subsidiary issue.

arbitrary homotopy level without a strict extensional equality coexisting beside the univalent homotopic equality. Indeed, simply extending ITT with univalence only gives the intensional (judgemental) subset of strict equality. Conversely, directly extending ETT with univalence so as to get a fully extensional strict equality leads to an inconsistent system. So, restrictions on the ability to rewrite from univalent equality to strict extensional equality have to be considered.

References

[1] Robin Adams. Pure type systems with judgemental equality. *J. Funct. Program.*, 16(2):219–246, 2006.

[2] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 412–420. IEEE Computer Society, 1999.

[3] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68, 2007.

[4] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The calculus of algebraic constructions. *CoRR*, abs/cs/0610063, 2006.

[5] Frédéric Blanqui, Jean-Pierre Jouannaud, and Pierre-Yves Strub. Building decision procedures in the calculus of inductive constructions. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, Lecture Notes in Computer Science, pages 328–342. Springer, 2007.

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

[7] Daniel R. Licata and Robert Harper. Canonicity for 2-dimensional type theory. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 337–348. ACM, 2012.

[8] Nicolas Oury. *Egalité et filtrage avec types dépendants dans le calcul des constructions inductives.* Phd thesis, University Paris 11, 2006.

[9] Vincent Siles and Hugo Herbelin. Pure type systems conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, Mar 2012.

[10] Pierre-Yves Strub. Coq modulo theory. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2010. `http://pierre-yves.strub.nu/coqmt/`.

[11] Vladimir Voevodsky. Univalent foundations of mathematics. In *Logic, Language, Information and Computation*, volume 6642 of *Lecture Notes in Computer Science*, page 4, Berlin - Heidelberg, 2011. Springer.

The Rooster and the Syntactic Bracket

Hugo Herbelin[1] and Arnaud Spiwack[1]

Inria Paris-Rocquencourt
Paris, France
hugo.herbelin@inria.fr, arnaud@spiwack.net

Awodey & Bauer [2] introduced *bracket types*. In an extensional type theory they show that every *squash type* – a type is squash if all its terms are equal – can be construed as being in the image of a *bracketing* operation. Given such an operation, they can prove that squash types contain first-order logic.

The main rule governing bracket types is the elimination rule (the type formation rule and the introduction rules are the same as those given below):

$$\frac{\Gamma \vdash q : [A] \quad \Gamma \vdash B : \mathrm{Type} \quad \Gamma, x{:}A \vdash b : B \quad \Gamma, x{:}A, y{:}A \vdash b = b[x \setminus y] : B}{\Gamma \vdash \text{match } q \text{ with } \langle x \rangle \Rightarrow b : B}$$

The side condition involving equality makes it apparently a construct of extensional type theory. However, we will show that the pattern-matching typing rules of Coq's sort Prop [1] can be modelled with a syntactic variant of this elimination rule for bracket types. Maybe more surprisingly, a slight change in the typing rules will allow us to model the impredicative sort Set as well.

1 Prop

We consider an intensional type theory with a sort Type and type connectives 1, 0, $A + B$, $\prod_{x:A} B$, and $\sum_{x:A} B$. To be complete with respect to Coq, we would need inductive and coinductive fixed points of (strictly positive) type families and equality, but they are irrelevant to this presentation.

Coq's Prop doesn't contain only squash types, however Coq is compatible with the assumption that it does. It is, in fact, a crucial property for extraction. This serves as a guiding principle to model Prop in our small type theory. We introduce a sort Prop – with $\mathrm{Prop} : \mathrm{Type}$. This new sort is stable under 1, 0, $\prod_{x:A} B$ (for A of any sort), and $\sum_{x:A} B$ (for $A : \mathrm{Prop}$); but crucially not $A + B$ nor $\sum_{x:A} B$ for $A : \mathrm{Type}$.

As in Coq, these proposition constructors have unrestricted pattern-matching rules. Coq's Prop also sports disjunction and existential connectives, their elimination rules, however, are limited to a co-domain of type Prop. To model this behaviour we introduce a type constructor $[A]$ whose typing rules are as follows:

$$\frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash [A] : \mathrm{Prop}}$$

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \langle x \rangle : [A]}$$

$$\frac{\Gamma \vdash q : [A] \quad \Gamma \vdash B : \mathrm{Prop} \quad \Gamma, x{:}A \vdash b : B}{\Gamma \vdash \text{match } q \text{ with } \langle x \rangle \Rightarrow b : B}$$

Thanks to these typing rules, we can define the disjunction as being $[A + B]$ and the existential quantifier to be $[\sum_{x:A} B]$. Coq's pattern matching rules can be derived from these definitions.

2 Set

Coq has an optional impredicative sort Set, it works as Prop except that it can be proven that it contains types which are not squash. We can, likewise, extend our small type theory with a sort Set stable under 1, 0, $\prod_{x:A} B$ (for A of any sort), $\sum_{x:A} B$ (for A : Set), *as well as* $A + B$. Equipped – *mutatis mutandis* with the bracketing operation, it models the appropriate restriction to the pattern matching of dependent sums.

References

[1] The Coq Proof Assistant.

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

Effective Types for C formalized in Coq

Robbert Krebbers

ICIS, Radboud University Nijmegen, The Netherlands

Abstract

The C11 standard describes dynamic typing restrictions (called *effective types*) to allow compilers to make more effective non-aliasing hypotheses. We present a formal version of effective types and prove that it satisfies some desirable properties. This work is part of an ongoing project to develop a formalization for a large part of C11 in Coq.

1 Introduction

Pointers play an important role in the C programming language. A difficult aspect of pointers in C is that they allow *aliasing*. That is, it is possible to have multiple pointers that point to the same location in memory. Consider:

```
int f(int *p, int *q) { int x = *q; *p = 10; return x; }
```

When the function `f` is called with the same integer pointer for the arguments `p` and `q`, the assignment to `*p` also affects `*q`. This means that a compiler is not allowed to transform the function's body into `*p = 10; return (*q);`. To restrict the situations in which aliasing may occur, the C standard introduced the notion of *effective types*. Consider:

```
float g(int *p, float *q) { float x = *q; *p = 10; return x; }
```

In this case, the function `g` has arguments of different types. Effective types allow a compiler to assume that `p` and `q` do not alias as their types differ, and thereby allow a compiler to reorder the function's body accordingly. When this function is called with aliased pointers[1]

```
union { int x; float y; } u = { .y = 3.14 }; g(&u.x, &u.y);
```

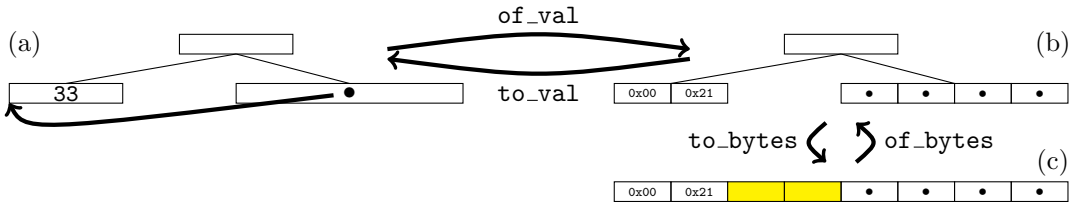
undefined behavior occurs. Undefined behavior occurs at run time (it cannot be checked for statically), and allows a program to do literally anything. This is to avoid compilers having to insert (expensive) dynamic checks to handle corner cases. A compiler thus does not have to test if effective types are violated (here: to test whether `p` and `q` are aliased), but is allowed to assume no violations occur. Not treating these undefined behaviors in a formal semantics makes it possible to prove certain programs to be correct whereas they may crash when compiled with an actual C compiler. Hence, a formal C semantics should take (effective) types seriously.

2 Approach

Significant existing versions of a formal semantics for C (*e.g.*: [1, 4]) describe the memory as a collection of cells, each of them consisting of an array of bytes. We take this approach further, and use well typed trees (called *memory values*) as cells. Pointers are represented as a pair of a cell identifier and a path through the corresponding memory value. Representing the memory as a typed forest allows us to keep track of the variants of unions internally, and thereby gives a natural semantics for effective types.

[1] A `union` is C's version of a *sum* type, and a `struct` of a *product* type. Unions are *untagged* instead of *tagged*, which means that the current variant of the union cannot be obtained unless explicitly stored.

The main novelty of our model is that the memory, although being fairly abstract, still supports low level operations; in particular byte-wise copying of objects. The model includes three layers: (a.) *values* with machine integers and pointers as leafs, (b.) *memory values* with arrays of bytes as leafs, and (c.) arrays of bytes. As in CompCert [4], bytes corresponding to uninitialized memory and fragments of pointers are symbolic. The following figure displays these three layers for the object `struct { short x, *p; } s = { 33; &s.x }`.



When copying an object by assignment, it is converted to an actual value when loaded (using `to_val`) and converted back when stored (using `of_val`). The conversion back yields a set of possible representations. When copying an object byte-wise (using `to_bytes` and `of_bytes`), information about the variants of contained unions may be lost (as that cannot be stored in bytes). We define a partial order $\subseteq$ to capture that information about the variants of unions may have gotten lost, and that uninitialized memory may have become initialized.

Theorem 2.1. *For each memory value v and type τ with $v : \tau$, there exists a memory value w such that: (a.) $w \in$* `of_val` *(*`to_val` *v), and (b.) $w \subseteq$* `of_bytes` *τ (*`to_bytes` *v).*

A consequence of the previous theorem is that a copy by assignment can be transformed into a byte-wise copy. If a copy of v by assignment has defined behavior, then it does not go wrong for each representation obtained by conversion to a value and back. As for one of these representations w we have $w \subseteq$ `of_bytes` τ (`to_bytes` v) by the theorem, and the fact that all operations on the memory respect $\subseteq$, we obtain soundness of this transformation.

3 Future work

The eventual goal of this work is to develop a semantics for a large part of the C11 programming language. In previous work [3] we developed a concise operational and axiomatic semantics for non local control flow (goto and return statements). Recently, we have extended this operational and axiomatic semantics with sequence points and non-deterministic expressions with side-effects. The next step is to create a fully fledged memory model on top of the abstract version introduced here, and to integrate it into our formalization [2]. Another direction for future research is to prove a correspondence with the CompCert memory model [4].

References

[1] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[2] Robbert Krebbers. The CH_2O formalization, 2013. `http://robbertkrebbers.nl/research/ch2o/`.

[3] Robbert Krebbers and Freek Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS*, volume 7794 of *LNCS*, pages 257–272, 2013.

[4] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, 2012.

Type-based Human-Computer Interaction

Peter Ljunglöf

Department of Computer Science and Engineering,
University of Gothenburg and Chalmers University of Technology, Sweden

A dialogue system is a computer system that can engage in dialogue with a human in order to complete tasks such as answering questions or performing actions. The information state update (ISU) approach [2] is a linguistically motivated theory of how to design dialogue systems. The approach is capable of advanced dialogue behaviour, and is based on a structured information state to keep track of dialogue context information.

The underlying logic of ISU-based systems tend to get very complicated, making it difficult to foresee the side effects of changing, adding or deleting inference rules. Ranta and Cooper [5] describe how a dialogue system can be implemented in a syntactical proof assistant based on type theory. Metavariables in the proof tree represent questions that needs to be answered by the user so that the system can calculate a final answer. However, the backbone of their theory is a very simple form-based dialogue system that cannot handle underspecified answers, anaphoric expressions, or ambiguous utterances.

In [3], we extended their theory with ideas from ISU and Dynamic Syntax [1], to make the system handle more flexible dialogues. Ranta and Cooper use *focus nodes* to know which of the nodes in the tree that is the current *question under discussion*. To this we add the idea of *unfixed nodes* from Dynamic Syntax. An unfixed node is a subtree which we know should be attached somewhere below a given node, but we do not yet know exactly where. We use unfixed nodes for representing *underspecified* answers, which is a very common phenomenon occuring in everyday dialogue between human participants.

In figure 1 the top node is the current focus, meaning that the system wants to build a tree of the type Action, which can be transliterated as the question "what do you want to do?". The user has given a partial answer to that question, by saying that (s)he wants to "go to London". This answer is of the type Route, which is not the correct type. Therefore the system adds this partial tree as an *unfixed* descendant of the focus node. The meaning is that in the final tree, the Action node will contain the Route tree as descendant.

We use a type-theoretical grammar formalism [4] to specify all aspects of the dialogue domain: tasks, issues, plans and forms, as well as the ontology of individuals, properties and predicates. We then make use of type checking for constraining the dialogue trees. Type checking is also used when interpreting user utterances and when providing the user with suggestions of what to say next.

The goal of the dialogue is to build a complete type-correct tree, and when this tree is completed, it represents a task which the user wants the system to perform in some way. In our theory the system tries to build the tree by successive refinement. In the middle of the dialogue, the uninstantiated parts of the tree are represented with metavariables. There is always exactly one of these metavariables that has focus.

The general idea of the dialogue manager is to ask the user to refine the current focus node. User utterances are then translated to (possibly incomplete) subtrees, which the system tries to incorporate into the dialogue tree. If the user utterance is of the same type as the focused metavariable, the tree can be extended directly. Otherwise the system will try to add the utterance as an unfixed node below the focus, or it will try to change focus to another metavariable that has a compatible type.

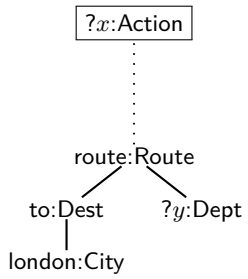


Figure 1: Example dialogue tree.

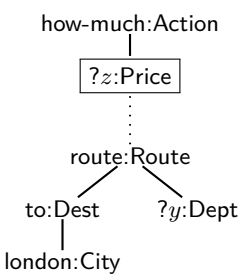


Figure 2: Refinement top-down.

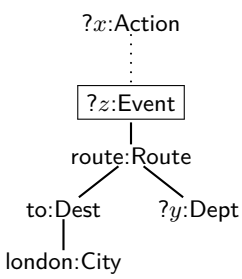


Figure 3: Refinement bottom-up.

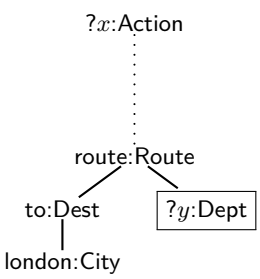


Figure 4: Refinement bottom-down.

There are at least three possible refinement strategies for deciding which node to select as the next focus. These strategies correspond to different dialogue behaviour:

- In *top-down* refinement (figure 2), the system tries to refine the focus node by finding possible children that can be ancestors to the unfixed trees.
- In *bottom-up* refinement (figure 3), the system tries to refine the unfixed node upwards by finding possible parents that can be descendants to the ancestor node.
- In *bottom-down* refinement (figure 4), the system tries to complete the unfixed tree first, and then finds its way up towards the ancestor node.

Issues that can be discussed within our proposed framework include the following:

- If the system is a question-answer (QA) system, the final complete tree specifies a question from the user, and the system can use type-theoretic function definitions for transforming the question into a suitable answer.
- Apart from unfixed nodes, we also borrow the idea of *linked trees* from Dynamic Syntax, and use them for sub-dialogues and anaphoric expressions.
- We have not yet incorporated dialogue feedback such as clarification questions and corrections into the theory. Feedback plays an importand role in dialogue, and it is necessary to be able to handle it in an advanced dialogue system.
- Ranta and Cooper borrowed the ideas of metavariables and focus nodes from the Agda proof assistant. Hopefully the new additions of unfixed nodes and refinement strategies can be borrowed back, and be used for making proof assistants more intuitive to use.

References

[1] Ruth Kempson, Wilfried Meyer-Viol, and Dov Gabbay. *Dynamic Syntax: The Flow of Language Understanding*. Blackwell, 2001.

[2] Staffan Larsson and David Traum. Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural Language Engineering*, 6(3–4):323–340, 2000.

[3] Peter Ljunglöf. Dialogue management as interactive tree building. In *DiaHolmia'09, 13th Workshop on the Semantics and Pragmatics of Dialogue*, Stockholm, Sweden, 2009.

[4] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011.

[5] Aarne Ranta and Robin Cooper. Dialogue systems as proof editors. *Journal of Logic, Language and Information*, 13(2):225–240, 2004.

Subtyping in Type Theory: Coercion Contexts and Local Coercions

Zhaohui Luo* and Fedor Part

Department of Computer Science
Royal Holloway, University of London
Egham, Surrey TW20 0EX, U.K.

The notion of subtyping is better understood for type assignment systems (those in programming languages) than for type theories with canonical objects (those in proof assistants) and, in this work, we are studying subtyping for the latter. As pointed out in [10, 8] among others, subsumptive subtyping, the traditional notion of subtyping with the subsumption rule, is incompatible with the notion of canonical object for inductive types in the sense that some key properties such as canonicity and subject reduction would fail to hold. It may be argued that coercive subtyping provides a more adequate alternative framework [10].

In this talk, we shall study two related constructs in coercive subtyping: coercion contexts and local coercions. These were introduced, and proved to be useful, in the context of employing type theories in linguistic semantics (see, for example, [9]). A *coercion context* is a context whose entries may be of the form $A <_c B$ as well as the usual form $x : A$:[1]

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : Type \quad \Gamma \vdash c : (A)B}{\Gamma,\ A <_c B\ \ valid} \qquad \frac{\Gamma,\ A <_c B,\ \Gamma'\ \ valid}{\Gamma,\ A <_c B,\ \Gamma' \vdash A <_c B : Type}$$

A *local coercion* is a subtyping assumption localised in terms (or judgements). For instance:

$$\frac{\Gamma,\ A <_c B \vdash k : K}{\Gamma \vdash (\mathbf{coercion}\ A <_c B\ \mathbf{in}\ k)\ :\ (\mathbf{coercion}\ A <_c B\ \mathbf{in}\ K)}$$

Note that the above constructs are the two sides of the same coin: subtyping relations can be assumed in a coercion context and they can be moved to the right of the $\vdash$-symbol to form terms with local coercions (otherwise, without local coercions, a subtyping entry in a context would block entries to its left from such moves[2]).

A formal treatment of coercion contexts and local coercions involves several technical issues. For example, validity of a coercion context is not enough anymore for it

*This work is partially supported by the grant F/07-537/AJ of the Leverhulme Trust in U.K.

[1] This is similar to coercion declarations in the proof assistants like Coq [3].

[2] In order to move subtyping assumptions to the right, one might consider an alternative approach: employing the so-called *bounded quantification* [1] that extends a higher-order calculus where quantification over (classes of) types are possible. However, the notion of bounded quantification is not well-understood and causes problems such as undecidability.

to be legal: instead, one needs to make sure that the context is *coherent*, guaranteeing the uniqueness of coercions. Also, the key word **coercion** distributes through the components of a judgement. For example, the conclusion judgement of the above rule should be identified with $\Gamma \vdash \mathbf{coercion}\ A <_c B\ \mathbf{in}\ (k : K)$. We shall give a formal presentation of coercion contexts and local coercions based on which a comparison to the coercive subtyping extension with global coercions [10] will be made. In particular, we shall prove that such an extension of type theory T with coercion contexts and local coercions is conservative in the sense that, if $\Gamma \vdash J$ is a judgement in T, then if $\Gamma \vdash J$ is derivable in the extension of T, it is derivable in T.

We shall also study the model-theoretic semantics of subtyping. Categorical semantics of dependent type theories have been studied (see, for example, [2, 5, 4] and more recently [7] for univalent foundations) and there has been research on models of subtyping for non-dependent type theories (see, for instance, [6]). We shall study categorical semantics of type theories with canonical objects extended by subtyping and, in particular, coercion contexts and local coercions.

This is work in progress.

References

[1] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17, 1985.

[2] J. Cartmell. Generalized algebraic theories and contextual category. *Annals of Pure and Applied Logic*, 32, 1986.

[3] Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.1), INRIA*, 2007.

[4] M. Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[5] M. Hyland and A. Pitts. The theory of constructions: categorical semantics and topos-theoretic models. *Categories in Computer Science and Logic, Boulder*, 1987.

[6] B. Jacobs. Subtypes and bounded quatification from a fibred perspective. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[7] C. Kapulkin, P. L. Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. Manuscript, 2012.

[8] Z. Luo. Notes on universes in type theory. Lecture notes for a talk at Institute for Advanced Study, Princeton (URL: `http://www.cs.rhul.ac.uk/home/zhaohui/universes.pdf`), 2012.

[9] Z. Luo. Formal semantics in modern type theories with coercive subtyping. *Linguistics and Philosophy*, 2013. (in press).

[10] Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: theory and implementation. *Information and Computation*, 223, 2013.

Probability Logics in Coq

Petar Maksimović[1,2]*

[1] Mathematical Institute of the Serbian Academy of Sciences and Arts, Belgrade, Serbia
[2] INRIA Sophia Antipolis Méditerranée, France
petarmax@mi.sanu.ac.rs

1 Introduction and Motivation

Probability logics have been viewed as tools for the estimation of uncertainty ever since the times of Leibnitz, Bernoulli and Boole, while the first one to provide probability theory with a model-theoretic approach was H. Jerome Keisler in [1], by introducing probability quantifiers of the form, for instance, $Px > r$, with the semantics that $(Px > r)\psi(x)$ means that the set $\{x \mid \psi(x)\}$ has probability greater than r. Another approach, where probabilities express degrees of belief, making it possible to capture propositions such as "The probability of an event is greater than x", was presented by Nils Nilsson in [2], has been further developed in papers such as [3, 4, 5], and that is the approach we will be adopting here.

Today, probability logics, in their various forms, constitute a framework for encoding probability-related statements, and offer a possibility and methodology for deriving conclusions from such statements, in a manner analogous to that of classical or intuitionistic logic, via axioms and inference rules. Probability logics, serving as decision-making or decision-support systems, can be a basis for expert systems used in economy, game theory, and medicine. As such, their correct functioning is of great importance, and formal verification of their properties appears as a natural step for one to take. In this short paper, we briefly describe the encoding and formal verification in Coq of the key properties of the probability logic $LPP_2^{\mathbb{Q}}$, which we present in the next Section. The structure of the obtained proofs in Coq has an outline that can be adapted and re-used for proving properties of other probability and modal-like logics.

2 $LPP_2^{\mathbb{Q}}$ in Coq

Due to space restrictions, we will present only an outline of the probability logic $LPP_2^{\mathbb{Q}}$ and several comments about its encoding and proofs of key meta-properties in Coq. $LPP_2^{\mathbb{Q}}$ is a modal-like logic obtained from classical propositional logic, by adding on top of it a layer of probabilistic operators of the form $P_{\geq r}$, where $r \in \mathbb{Q}_{[0,1]}$, and probabilistic formulas of the form $P_{\geq r}\alpha$, where α is a classical formula. Therefore, it would be possible to have statements such as $P_{\geq 0.3}\alpha \wedge P_{\geq 0.7}\beta$, with the meaning "The probability of α being true is at least 0.3 and the probability of β being true is at least 0.7". However, combinations of formulas with probabilistic and classical formulas, such as $(P_{\geq 0.1}\alpha) \rightarrow \beta$, or iterations of probability operators, such as $P_{\geq 0.1}(P_{\geq 0.9}\alpha)$, are not allowed. The semantics of $LPP_2^{\mathbb{Q}}$ are based on a possible-world approach, in which probability is captured by using a measure whose co-domain is $\mathbb{Q}_{[0,1]}$. The axiom system of $LPP_2^{\mathbb{Q}}$ contains the axioms and inference rules of classical propositional logic, plus axioms which capture probability, such as:

$$P_{\leq r}\alpha \rightarrow (P_{<s}\beta \rightarrow P_{<r+s}(\alpha \vee_c \beta)), \text{ for } r + s \leq 1,$$

*The research work presented in this paper has been supported by the Serbian Ministry of Education, Science and Technological Development, projects III44006 and ON174026.

and two new inference rules, one of which is infinitary (depends on countably many hypotheses):

Probabilistic Necessitation: from α, infer $P_{\geq 1}\alpha$.

Domain Enforcement: from $P_{\neq r}\alpha$, for all $r \in \mathbb{Q}_{[0,1]}$, infer $\bot$.

The notions of satisfiability, validity, proofs, derivability, theorems, and consistency are defined, and the following main theorems have been proven:

Soundness : If a formula F is a theorem, then F is valid.

Strong Completeness : A set of formulas T is consistent *if and only if* it is satisfiable.

Non-compactness: There exists a set of formulas T, such that its every finite subset is satisfiable, but the entire set is not satisfiable.

The proof of strong completeness, consisting of extending a consistent set of formulas to a maximally consistent set, and then constructing a canonical model, can be, with relative ease, adapted and re-used for other probability and modal-like logics. The formalization of $LPP_2^{\mathbb{Q}}$ in Coq, despite not including the encoding of binders and quantifiers (as $LPP_2^{\mathbb{Q}}$ is propositional in nature), has several interesting particularities, such as the encoding of the mentioned infinitary inference rule (similar to [6], the encoding of the models, the construction of an infinite union of expanding sets and reasoning over it, and the construction of a counter-example for compactness. For comparison, the reader can refer to [6, 7].

3 Further Work

The formal verification procedure described here has already been applied to three other probability logics similar to $LPP_2^{\mathbb{Q}}$, two with allowed iterations of probability operators, and two with a finite measure co-domain. From here, there are two main research directions which could be investigated. The first one is to find and/or design other probabilistic or modal-like logics (such as temporal logics) to which we could apply this procedure. The second direction is to specify in Coq a decision procedure for one of the formally verified logics, and extract a certified probabilistic SAT-checker from this specification.

References

[1] H. Jerome Keisler. Hyperfinite model theory. In *Gandy and Hylan, eds. Logic Colloquium '76*, pages 5–110. North Holland, Amsterdam, 1977.

[2] Nils J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71 – 87, 1986.

[3] Z Ognjanović and M Rašković. Some probability logics with new types of probability operators. *Journal of Logic and Computation*, 9(2):181–195, 1999.

[4] Zoran Ognjanović and Miodrag Rašković. Some first-order probability logics. *Theoretical Computer Science*, 247(1 - 2):191 – 212, 2000.

[5] Zoran Marković, Zoran Ognjanović, and Miodrag Rašković. A probabilistic extension of intuitionistic logic. *Mathematical Logic Quarterly*, 49(4):415–424, 2003.

[6] Furio Honsell and Marino Miculan. A natural deduction approach to dynamic logic. In Stefano Berardi and Mario Coppo, editors, *TYPES*, volume 1158 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 1995.

[7] P. de Wind. Modal logic in Coq. Master's thesis, Faculty of Exact Sciences, Vrije Universiteit in Amsterdam, 2011.

A Dependent Delimited Continuation Monad

Pierre-Marie Pédrot[1]

PPS, πr^2 team, Univ. Paris Diderot, Sorbonne Paris Cité,
UMR 7126 CNRS, INRIA Paris-Rocquencourt, Paris, France
pierre-marie.pedrot@inria.fr

Delimited continuations are a refinement of usual control operators, where one actually cares about the return type of an expression, for continuations do return. They provide us with a lot of advantages, the most remarkable one being the eponymous operation known as *delimitation*. In direct style, delimitation can be thought as being able to actually perform the effects lurking within the computation and obtain a pure value, while in indirect style, it comes embodied by a `run` operator allowing us to escape the monad. On the logical side, such a primitive permits to construct realizers of interesting semi-classical formulae [5], which are computable but not intuitionistically provable, while on a more programmatic side, it bears the ability to internalize any computational monad [3].

While non-delimited control naturally dwells in a call-by-name setting, delimited continuations are better behaved with respect to call-by-value calculi, because of the very existence of a notion of *values* and thence the sharp behavioral distinction arising between positive and negative types [9]. As for non-delimited control, things even get worse when one tries to grasp the utmost feature of positive types, namely *dependent elimination* [2]. Alas, it is quite well-known indeed that those two do not mix well at all [4].

We believe in dependent elimination being an invaluable asset for the slightest expressive logical system. Therefore, we consider than we cannot afford to forego the expressivity of dependent types in exchange for a more potent but degenerate classical logic. To recover the best of both worlds, we propose a generalization of the delimited continuation monad in a dependently typed setting, based upon the wise teachings of polarization.

Our monad can be seen as an instance of the usual parameterised monad used to encode delimited control [1], where we do not only track the return type of the continuation, but also the type of the whole potentially dependent stack of arguments computed until then. Such a structure is better known as a *telescope* in dependent type theory. The resulting object is of the following shape:

$$(\Pi(x_1 : A_1).\, \Pi(x_2 : A_2\, x_1).\, \Pi(x_3 : A_3\, x_1\, x_2).\, \ldots\, I\, x_1\, \ldots\, x_n) \Rightarrow O$$

In this monad, usual operators are granted finer types describing accurately their computational behavior. Going back and forth through the reification theorem of delimited continuations, this approach can also be applied to a whole range of monads.

Thus we can capture a common design pattern in a dependently typed world without much fuss. This may prove valuable to any touchy programmer willing to simulate impure effects in Coq and transpose her Haskell knowledge to a richer type system, for example.

Thanks to dependent elimination, we also get interesting logical features of this structure. For instance, we are able to obtain something akin to invertibility properties for negative connectives from linear logic. Careful insight also suggests that we may want to relax the telescope definition to allow co-inductive and infinitary negative types, to cope with certain defects of intuitionistic logic.

In this talk, we will recall the basis of delimited continuations and dependent elimination in order to present our monad. We will explain how it naturally merges the two settings, and

show how it refines the type of famous operators, as well as some nice offshoots we get from this unexpected marriage.

References

[1] Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, July 2009.

[2] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[3] Andrzej Filinski. Representing monads. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457. ACM Press, 1994.

[4] Hugo Herbelin. On the degeneracy of sigma-types in presence of computational classical logic. In *Typed Lambda Calculi and Applications, 7th International Conference*, pages 209–220, 2005.

[5] Danko Ilik. Delimited control operators prove Double-negation Shift. *Annals of Pure and Applied Logic*, 163(11):1549–1559, November 2012.

[6] Yukiyoshi Kameyama. Axioms for control operators in the CPS hierarchy. *Higher Order Symbol. Comput.*, 20(4):339–369, December 2007.

[7] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely: System description. In *Functional and Logic Programming, 10th International Symposium*, pages 304–320, 2010.

[8] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[9] Noam Zeilberger. Polarity and the Logic of Delimited Continuations. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science*, pages 219 – 227, Edinburgh, United Kingdom, 2010.

Type-theoretical natural language semantics: on the system F for meaning assembly

Christian Retoré*

Équipe Melodi, IRIT-CNRS, UPS, 118 route de Narbonne, 31062 Toulouse Cedex 9
(& LaBRI, Université de Bordeaux, 351 cours de la Libération, 33405 Talence cedex)

Roughly speaking, the standard semantical analysis of natural language consists in mapping a sentence $s = w_1 \cdots w_n$ to a logical formula $[\![s]\!]$ which depicts its meaning. It is a computational process which implements Frege's compositionally principle. A parser turns the sentence s into a binary parse tree t_s specifying at each node which subtree is the function — the other subtree being its argument. The lexicon provides each leaf of t_s, that is a word w_i, with a λ-term $[\![w_i]\!]$ over the base types t (propositions) and e (individuals). By structural induction on t_s, we obtain a λ-term $[s]{:}\mathsf{t}$ corresponding to t_s. Its normal form, that is a formula of higher order logic, is $[\![s]\!]{:}\mathsf{t}$, the meaning of s. This standard process at the heart of Montague semantics relies on Church's representation of formulae as simply typed λ-terms, see e.g [7, Chapter 3].

It would be more accurate to have many individual base types rather than just e. This way, the application of a predicate to an argument may only happen when it makes sense. For instance sentences like *"The chair barks."* or *"Their five is running."* are easily ruled out when there are several types for individuals by saying that *"barks"* and *"is running"* apply to individuals of type *"animal"*. Nevertheless, such a type system needs to incorporate some flexibility. Indeed, in the context of a football match, the second sentence makes sense, because *"their five"* may be understood as a player who, being *"human"*, is an *"animal"* that can run.

These meaning transfers have been receiving much attention since the 80s, as [1] shows. As [1, 5], we too proposed a formal and computational account of these phenomena, based on Girard's system F (1971) [3]. We explored the compositional properties (quantifiers, plurals and generic elements,....) as well as the lexical issues (meaning transfers, copredication, fictive motion,...) [2, 8, 9]. Our system works as follows: the lexicon provides each word with a main λ-term, the "usual one" which specifies the argument structure of the word, by using refined types: *"runs:* $\lambda x^{animal}\underline{run}(x)$*"* only applies to *"animal"* individuals. In addition, the lexicon may endow each word with a finite number of λ-terms (possibly none) that implement meaning transfers. For instance a *"town"* may be turned into an *"institution"*, a geographic *"place"*, or a football *"club"* by the optional λ-terms *"f_i: town→institution"*, *"f_p: town → place"* and *"f_c: town → club"* — in subtler cases these λ-terms may be more complex than simple constants. Thus, a sentence like *"Liverpool is a large harbour and decided to build new docks."* can be properly analysed. Some meaning transfers, like f_c, are declared to be *rigid* in the lexicon. Rigidity prohibits the simultaneous use of other meaning transfers. For instance, the rigidity of f_c properly blocks *"* Liverpool defeated Chelsea and decided to build new docks."*.

The polymorphism of system F is a welcome simplification. For instance, a single type $\Pi\alpha.(\alpha \to \mathsf{t}) \to \mathsf{t}$ is enough for the quantifiers $\forall$ or $\exists$x. Polymorphism also allows a factorised treatment of conjunction for copredication: *whenever* an object x of type ξ can be viewed both as an object of type α to which a property $P^{\alpha\to\mathsf{t}}$ applies and as an object of type β to which a property $Q^{\beta\to\mathsf{t}}$ applies (via two optional terms $f_0 : \xi \to \alpha$ and $g_0 : \xi \to \beta$), x enjoys $P \wedge Q$ can be expressed with $\Lambda\alpha\Lambda\beta\lambda P^{\alpha\to\mathsf{t}}\lambda Q^{\beta\to\mathsf{t}}\Lambda\xi\lambda x^{\xi}\lambda f^{\xi\to\alpha}\lambda g^{\xi\to\beta}.\ (\wedge^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}\ (P\ (f\ x))(Q\ (g\ x)))$, i.e., with a *single* polymorphic *"and"*. Our logical system also has two layers that slightly differ from

*This research was supported by the ANR projects Loci and Polymnie.

Montague's. Our *meta logic* (a.k.a. glue logic) is system F (instead of simply typed λ-calculus) with base types t, $(\mathsf{e}_i)_{i\in I}$ (instead of a single type e) Our *logic for semantic representations* is many-sorted higher-order logic — the e_i being the sorts. Quantifiers are preferably represented by Hilbert's operators, that are constants $\epsilon, \tau : \Lambda\alpha.\ (\alpha \to \mathsf{t}) \to \alpha$ [8]. An easy but important property holds: if the constants define an n-order q-sorted logic, any (η-long) normal λ-term of type t corresponds to a formula of n-order q-sorted logic (possibly $n = \omega$).

We preferred system F to modern type theories (MTT) of [4] and to the categorical logic of [1] because of its formal simplicity and its absence of variants. Furthermore, F terms with a problematic complexity are avoided, since semantical terms derive from the simple terms in the lexicon by means of simple syntactic rules. Nevertheless there are two properties of [4] that are welcome: a proper notion of subtyping, mathematically safe and linguistically relevant, and predefined inductive types with specific reduction rules. Indeed, subtyping naturally represents ontological inclusions (a *"human being"* is an *"animal"*, hence predicates that apply to *"animals"* also apply to *"human beings"*). Coercive subtyping [11] sounds promising for F. Its key property is that, if at most one subtyping map is given between any two base types, then there also is at most one subtyping map between any two complex types. Predefined (inductive) types, e.g. integers as in Gödel's system T and finite sets of α-objects with their reduction schemes as in [10] are also welcome — encodings in F are cumbersome. The key points are normalisation, confluence and the absence of closed constant-free terms in any false type. We shall also illustrate the linguistic relevance of these extensions, which are already included in Moot's semantical and semantical parser for French named Grail. [6]

References

[1] Nicholas Asher. *Lexical Meaning in context – a web of words.* Cambridge University press, 2011.

[2] Christian Bassac, Bruno Mery, and Christian Retoré. Towards a Type-Theoretical Account of Lexical Semantics. *Journal of Logic Language and Information*, 19(2):229–245, April 2010. `http://hal.inria.fr/inria-00408308/`.

[3] Jean-Yves Girard. *The blind spot – lectures on logic.* European Mathematical Society, 2011.

[4] Zhaohui Luo. Contextual analysis of word meanings in type-theoretical semantics. In Sylvain Pogodalla and Jean-Philippe Prost, editors, *LACL*, volume 6736 of *LNCS*, pages 159–174. Springer, 2011.

[5] Zhaohui Luo. Common nouns as types. In Denis Béchet and Alexander Ja. Dikovsky, editors, *LACL*, volume 7351 of *Lecture Notes in Computer Science*, pages 173–185. Springer, 2012.

[6] Richard Moot. Wide-coverage French syntax and semantics using Grail. In *Proceedings of Traitement Automatique des Langues Naturelles (TALN)*, Montreal, 2010.

[7] Richard Moot and Christian Retoré. *The logic of categorial grammars: a deductive account of natural language syntax and semantics*, volume 6850 of *LNCS*. Springer, 2012. `http://www.springer.com/computer/theoretical+computer+science/book/978-3-642-31554-1`.

[8] Christian Retoré. Variable types for meaning assembly: a logical syntax for generic noun phrases introduced by "most". *Recherches Linguistiques de Vincennes*, 41:83–102, 2012. `http://hal.archives-ouvertes.fr/hal-00677312`.

[9] Christian Retoré. Sémantique des déterminants dans un cadre richement typé. In Emmanuel Morin and Yannick Estève, editors, *Traitement Automatique du Langage Naturel, TALN 2013*, pages 1–14. ACL Anthology, 2013.

[10] Sergei Soloviev and David Chemouil. Some Algebraic Structures in Lambda-Calculus with Inductive Types. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 338–354. Springer, 2003.

[11] Sergei Soloviev and Zhaohui Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 1-3(113):297–322, 2000.

Sets in homotopy type theory

Egbert Rijke and Bas Spitters *

Radboud University Nijmegen

Our paper contributes to the univalent homotopy type theory [4, 2, 7] program which connects and extends a number of topics such as type theory, (elementary) topos theory, homotopy theory, higher category theory and higher topos theory. It envisions a natural extension of the Curry-Howard correspondence between simply typed λ-calculus, Cartesian closed categories and minimal logic, via extensional dependent type theory, locally Cartesian closed categories and predicate logic all the way to univalent homotopy type theory (HoTT) and higher toposes. A key missing ingredient is a first-order ('elementary') definition of a higher topos. It has been conjectured [2, 6] that homotopy type theory including the univalence axiom and higher inductive types can be used as such an internal language, and thus may provide an elementary definition of a higher topos. Another missing ingredient is the lack of a computational interpretation for the univalence axiom.

We connect sets (0-types) with predicative toposes [5, 3]. The prototypical example of a predicative topos is the category of setoids in Martin-Löf type theory. A setoid is a groupoid in which all hom-sets have at most one inhabitant. A further generalization of setoids is to ∞-groupoids. Grothendieck conjectured that Kan simplicial sets and ∞-groupoids are equivalent, however, precisely defining this equivalence is the topic of active research around the 'Grothendieck homotopy hypothesis'. Both the 0-truncated groupoids and the 0-truncated Kan fibrations are similar to setoids, and constructively so. This is made precise for instance by the fact that the 0-truncation of a model topos is a Grothendieck topos [8, Prop 9.2] and every Grothendieck topos arises in this way [8, Prop 9.4]. We prove an internal version of the former result.

Theorem 1. *The category* Set *is a* ΠW*-pretopos.*

An internal version of the latter result may follow by carrying out Coquand's proposed constructive Kan fibrations in the predicative topos of sets [1].

Predicative topos theory follows the methodology of Algebraic Set theory, a categorical treatment of set theory, which in particular captures the notion of smallness by considering maps with appear as a pullback of a universally small map. It extends the ideas from the elementary theory of the category of sets (ETCS) by including universes. Using the univalence axiom, we show that $\mathcal{U}$ behaves like the object classifier from higher topos theory.

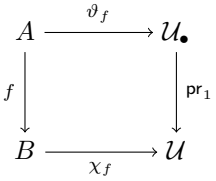
Theorem 2. *For every type* B *there is an equivalence*

$$\chi : \left(\sum (A : \mathcal{U}),\ A \to B \right) \simeq B \to \mathcal{U}$$

given by $\chi(A, f) :\equiv \lambda b.\mathsf{hFiber}(f, b)$*. Moreover, for any* $f : A \to B$ *the function* $\chi(A, f) : B \to \mathcal{U}$

*The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

is the unique function which fits in a pullback diagram



Here $\mathcal{U}_\bullet$ *is the type* $\sum(X : \mathcal{U}),\ X$ *of pointed types.*

This result remains true when we replace $\mathcal{U}$ by the type n-$\mathcal{U}$ of all types of homotopy level n. In particular we get

Theorem 3. *The projection* $\pi : (0\text{-}\mathcal{U})_\bullet \to 0\text{-}\mathcal{U}$ *is a universal small map for sets.*

Note, however, that not only 0-$\mathcal{U}$ is a large type, it is also not a set. Instead, 0-$\mathcal{U}$ is a 1-groupoid (by univalence). The class of pullbacks of π forms a stable class which is locally full. However, the collection axiom (and hence the axiom of multiple choice) do not seem to be derivable. We make a preliminary conjecture that higher inductive types can replace these axioms in many applications.

References

[1] B. Barras, T. Coquand, and S. Huber. A generalization of Takeuti-Gandy interpretation. 2013.

[2] S. Awodey. Type theory and homotopy. *Epistemology versus Ontology*, pages 183–201, 2012.

[3] B. van den Berg. Predicative toposes. *To appear*, 2012.

[4] C. Kapulkin, P. Lumsdaine, and V. Voevodsky. Univalence in simplicial sets. *http://arxiv.org/abs/1203.2553v2*, 2012.

[5] I. Moerdijk and E. Palmgren. Type theories, toposes and constructive set theory: predicative aspects of AST. *Ann. Pure Appl. Logic*, 114(1-3):155–201, 2002. Commemorative Symposium Dedicated to Anne S. Troelstra (Noordwijkerhout, 1999).

[6] M.Shulman. The univalence axiom for inverse diagrams. *arXiv:1203.3253*, 2012.

[7] Á. Pelayo, V. Voevodsky, and M. Warren. A preliminary univalent formalization of the p-adic numbers. *arXiv:1302.1207*, 2013.

[8] C. Rezk. Toposes and homotopy toposes (version 0.15). 2010.

Polymorphic variants in dependent type theory

Claudio Sacerdoti Coen[1*] and Dominic P. Mulligan[1]

Dipartimento di Informatica – Scienza e Ingegneria,
Università di Bologna,
Mura Anteo Zamboni 7, Bologna (BO) Italy

The expression problem and polymorphic variants The expression problem is maybe the best known issue in programming language development and concerns how best to extend a data type of expressions in order to include new constructors, with all operations defined on the data type being updated to cover the new constructors. Further, this extension should be modular: it should be possible to add new constructors, or to merge together two sets of constructors, without modifying the previously developed code. Object oriented languages provide a neat solution to the problem: the type of expressions becomes an interface that is implemented by a class representing each constructor. However, object oriented languages make a trade-off in sacrificing a closed universe of expressions, foregoing pattern matching and exhaustivity checking. Pattern matching, in particular, seems the natural way to reason on (expression) trees.

A partial solution can be obtained using functional languages based on algebraic types and pattern matching, like OCaml or Haskell. The idea consists of "opening" the algebraic data type E of expressions by turning it into a parametric type $E\ \alpha$ where the type parameter α replaces E in the recursive arguments of the constructors of E. The recursive type $\mu\alpha.(E\ \alpha)$ is then isomorphic to the original 'closed' type E. In order to merge together two types of expressions $E_1\ \alpha$ and $E_2\ \alpha$ it is sufficient to build the parametric disjoint union $E\ \alpha := K_1 : E_1\ \alpha \mid K_2 : E_2\ \alpha$. Similarly, given two functions f_1, f_2 typed as $f_i : (\alpha \to \beta) \to E_i\ \alpha \to E_i\ \beta)$, it is possible to build the function f over $E\ \alpha$ by pattern matching over the argument and dispatching the computation to the appropriate f_i.

The previous solution has several major problems. The first one is the non-associativity of the binary merge operation, which is a major hindrance to modularity. Concretely, it is often the case that one needs to explicitly provide functions to convert back and forth between isomorphic types. A second problem is the following: merge is implemented on top of a *disjoint* union, when the expected operation is the standard union. Again, this is a problem for modularity, since it prohibits multiple inheritance. Lastly, the solution is inefficient as every merge operation adds an indirection that costs both in space (memory consumption) and pattern matching time.

A satisfactory solution to the expression problem for functional languages is given by *polymorphic variants*, like the ones implemented by Guarrigue in the OCaml language [1, 2] . The idea is to add to the programming language a (partial) merge operation over algebraic types that corresponds to a standard union. Merging fails when the the same constructor occurs in both types to be merged with incompatible types. Reworking the previous construction with this operation already gives a satisfactory solution by solving at once all previous problems. Moreover, polymorphic variants and their duals, polymorphic records, allow for an interesting typing discipline where polymorphisms is obtained not by subtyping, but by uniformity. For example, a function could be typed as $[K_1 : T_1 \mid K_2 : T_2] \cup \rho \to T$ to mean that the input can

*The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

be any type obtained my merging a type ρ into the type of the two listed constructors. The function can be applied to a $K_3\ M$ by instantiating (via unification in ML) ρ with $[K_3 : T_3] \cup \sigma$ for some σ and for $M : T_3$.

An efficient encoding in dependent type theory In the talk we will demonstrate an *efficient* encoding of *bounded polymorphic variants* in a dependent type theory augmented with implicit coercions. The languages of Coq and Matita, for instance, can be used for the encoding, doing everything at user level. We use 'bounded polymorphic variants' for the class of all polymorphic variants whose constructors are a subset of one (or more) sets of typed constructors—called universes—given in advance.

Several encodings are possible. However, we will limit ourselves to the one that respects the following requirements:

1. Universe extension: after adding a new constructor to a universe, all files that used polymorphic variants that were subsets of the universe should typecheck without modification. Therefore, the restriction to the bounded case is not a problem as the merge of two universes does not require any changes to the code.

2. Efficiency of extracted code: after code extraction, bounded polymorphic variants and classical algebraic types should be represented in the same way and have the same efficiency.

3. Expressivity: all typing rules for polymorphic variants discusses in the literature must be derivable. In particular, each bounded polymorphic variant should come with its own elimination principle that allows one to reason only on the constructors of the universe that occur in the polymorphic variant.

4. Non-intrusivity: thanks to implicit coercions and derived typing rules, writing code that uses polymorphic variants should not require more code than what is typically required in OCaml.

Our encoding is based on the following idea: a universe is represented as a standard algebraic data type; a polymorphic variant on that universe is represented as a pair of lists of constructors, those that may and those that must be present; dependent types and computable type formers allow one to turn the two lists into a Σ-type of inhabitants of the universe that are built only from constructors that respect the constraints; code extraction turns the Σ-type into the universe type, ensuring efficiency; implicit coercions are used to hide the Σ-type construction, so that the user only works with the two lists; more dependently-typed type formers compute the type of the introduction and elimination rules for the polymorphic variants; the latter are inhabited by dependently typed functions. All previous functions and type formers cannot be expressed in the type theory itself. However, we provide a uniform construction (at the meta-level) to write them for each universe.

Lastly, we will show how the same ideas can be exploited for a similar efficient representation of polymorphic records in dependent type theory.

References

[1] Jacques Garrigue. Programming with polymorphic variants. In *ML workshop*, 1998.

[2] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on foundations of software engineering*, 2000.

Viewing λ-terms through Maps

Masahiko Sato[1] *, Randy Pollack[2] †, Helmut Schwichtenberg[3] and Takafumi Sakurai[4] ‡

[1] Graduate School of Informatics, Kyoto University
[2] School of Engineering and Applied Sciences, Harvard University
[3] Mathematics Institute, University of Munich
[4] Department of Mathematics and Informatics, Chiba University

In this paper we introduce the notion of *map*, using binary trees over 0 and 1. A map is a notation for *occurrence* of a symbol in syntactic expressions such as formulas or λ-terms. For example, consider a λ-term $(xz)(yz)$ in which each of the symbols x and y occur once and the symbol z occurs twice; we use (10)(00), (00)(10) and (01)(01) to represent the occurrences of the symbols x, y and z respectively. The bound positions are represented only by a distinguished constant $\Box$ (called *box*). We will write (omitting some parentheses for readability)

$$(10\ 00)\backslash(00\ 10)\backslash(01\ 01)\backslash(\Box\Box\ \Box\Box)$$

for the S combinator $\lambda xyz.\,(xz)(yz)$. $\Box$ may also occur unbound. Free variables may still occur in terms, e.g. the informal term $\lambda z.(xz)$ is written as $01\backslash(x\ \Box)$, but there are no bound names or de Bruijn indices.

Some well-formedness conditions will be required. Since we want a *canonical* representation (one notation per lambda term), although $\lambda x\lambda x.\,x$ is accepted as a correct notation for a lambda term, we will not accept $1\backslash1\backslash\Box$ as a correct notation. Also, consider the substitution of $(\Box\ \Box)$ for the position 10 (the first $\Box$) in $01\backslash(\Box\ \Box)$; we get $01\backslash((\Box\ \Box)\ \Box)$ which does not match the intuition hinted at above because 0 is not a position in $(\Box\ \Box)$. For this reason we identify the map $(0\ 0)$ with the map 0. (This notation appears already in [4]; in the paper we prove it correct.)

We develop a representation of lambda terms using maps. The representation is concrete (inductively definable in HOL or Constructive Type Theory) and canonical (one representative per λ-term). We define substitution for our map representation, and prove the representation is in substitution preserving isomorphism with both nominal logic λ-terms [3, 6] and de Bruijn nameless terms [1]. These proofs are mechanically checked in Isabelle/HOL/Nominal [6] and Minlog [5] respectively.

The map representation has good properties. Substitution does not require adjustment of binding information: neither α-conversion of the body being substituted into, nor de Bruijn lifting of the term being implanted. We have a natural proof of the substitution lemma of λ calculus that requires no fresh names or index manipulation.[1]

Using the notion of map we study conventional raw λ syntax. E.g. we give, and prove correct, a decision procedure for α-equivalence of raw λ terms that does not require fresh names.[2]

We conclude with a definition of β-reduction of map terms, some discussion on the limitations of our current work, and suggestions for future work.

*Supported by JSPS KAKENHI Grant Number 22300008.

†Supported by the DARPA CRASH program through the US Air Force Research Laboratory under Contract No. FA8650- 10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

‡Supported by JSPS KAKENHI Grant Number 24650002.

[1] See, e.g. [6] where the need for fresh names in this proof is discussed.

[2] See [2] for discussion of this problem.

Three abstraction mechanisms In this paper we study three abstraction mechanisms and the three associated representations of lambda terms: Λ of *raw λ-terms*, $\mathbb{L}$ of *map λ-expressions* and $\mathbb{D}$ of *de Bruijn-expressions*, generated by the following grammar:

$$K, L \in \Lambda ::= x \mid \Box \mid \mathsf{app}(K, L) \mid \mathsf{lam}(x, K).$$
$$M, N \in \mathbb{L} ::= x \mid \Box \mid \mathsf{app}(M, N) \mid \mathsf{mask}(m, M) \quad (m \mid M).$$
$$D, E \in \mathbb{D} ::= x \mid \Box \mid \mathsf{app}(D, E) \mid i \mid \mathsf{bind}(D).$$
$$x \in \mathbb{X} \quad \text{The type of } \textit{atoms} \text{ or } \textit{parameters}.$$
$$i \in \mathbb{I} \quad \text{The type of } \textit{natural numbers}, \text{ used as indices.}$$
$$m \in \mathbb{M} \quad \text{The type of } \textit{maps}.$$

The three abstraction mechanisms of these three representations are:

Lambda-abstraction Abstraction by *parameters*, realized by the constructor lam (λ) in Λ. Quotienting by α-equivalence is needed to make this mechanism work. The information about binding is shared between binding occurrences and bound occurrences (as shared names), and substitution may require adjusting both binding occurrences and bound occurrences of the base term (α-conversion).

Mask-abstraction Abstraction by *maps*, realized by the constructor mask in $\mathbb{L}$. (We write $m\backslash M$ for $\mathsf{mask}(m, M)$.) For this representation to work, mask must be guarded by a simultaneously defined relation, written $m \mid M$. Informally, $m \mid M$ means that M is well formed and m is a map of free boxes in M; thus $m\backslash M$ is itself well formed. $0 \mid M$ can be read as "M is well formed." The information about binding is carried only at the binding occurrences (as maps). Substitution does not require adjustment of binding information.

Bind-abstraction Abstraction by *indices*, realized by the constructor bind in $\mathbb{D}$. The information about binding is carried only at the bound occurrences (as indices pointing to the binding point). Substitution requires adjustment of the implanted term (de Bruijn lifting).

The three types, $\Lambda, \mathbb{L}$ and $\mathbb{D}$ all have *parameters* $x \in \mathbb{X}$ and $\Box$ (*box* or *hole*) as atomic objects, and have the constructor app in common. We compare these three types through maps. The main results of the paper are the machine checked proofs of adequacy of our representation.

References

[1] de Bruijn, N.G., Lambda calculus notation withe nameless dummies, a tool for automatic formal manipulation with application to the Church-Rosser theorem, Indag. Math., **34**, 381 – 392, 1972.

[2] Hendriks, D. and van Oostrom, V., Adbmal. In *Proceedings of the 19th Conference on Automated Deduction (CADE 19)* LNAI 2741, Springer-Verlag, 2003.

[3] Pitts, A., Nominal logic, a first order theory of names and binding, Information and Computation, **186**, pp. 165–193, 2003.

[4] Sato, M. and M. Hagiya, Hyperlisp, *Proceedings of the International Symposium on Algorithmic Language*, 251-269, North-Holland, 1981.

[5] Schwichtenberg, H., Minlog, in *The Seventeen Provers of the World* LNAI 3600, Springer-Verlag, 2006.

[6] Urban, C., Nominal Techniques in Isabelle/HOL, Journal of Automatic Reasoning, **40** (4), 2008.

Positive Logic Is 2-Exptime Hard

Aleksy Schubert, Paweł Urzyczyn, Daria Walukiewicz-Chrząszcz

Institute of Informatics, University of Warsaw
[daria,alx,urzy]@mimuw.edu.pl

Abstract

We prove that the positive fragment of minimal first-order intuitionistic logic is hard for doubly exponential time. The proof is based on an automata-theoretic characterisation.

We investigate the complexity of the positive fragment of (universally-implicational) first-order intuitionistic logic. Here, "positive" means that all universal quantifiers are located at positive positions in a formula. More precisely:

- The position of $\forall x$ in $\forall x\,\varphi$ is positive;
- Positive and negative positions in φ remain positive in $\forall x\,\varphi$ and in $\psi \to \varphi$.
- Positive and negative positions in ψ are respectively negative and positive in $\psi \to \varphi$.

Classically, such formulas are equivalent to universal (Π_1) formulas, and therefore have no more expressive power than quantifier-free predicate logic. However, intuitionistic logic is in general harder than classical. In particular, the equivalence between positive and Π_1 formulas is no longer valid, and the decision problem for the universal fragment is no longer obvious. Decidability for this fragment is due to Grigori Mints [5]. Later, Gilles Dowek and Ying Jiang [2] as well as Ivar Rummelhoff [6] gave independent, more comprehensive proofs, based on the Curry-Howard isomorphism. Yet another proof, by a reduction to the emptiness problem for context free grammars, is implicit in [3].

The decision algorithms given by these proofs are not easy. The upper bound obtained is not elementary, because every level of nested quantification yields an additional exponent. It remains an open question whether this upper bound can be improved. However, as we show below, the inherent complexity (and thus also the expressive power) of positive quantification is enormous anyway. We prove that the problem is hard for doubly exponential time.

This upper bound is obtained by an analysis of an appropriate machine model. Our *Eden automata* (or "expansible tree automata") are alternating machines operating on data structured into *trees of knowledge*. For a given automaton, the depth of the tree is bounded by a constant, but the width (the outdegree of nodes) is potentially unbounded. There is a fixed number of binary registers at every node of the tree. The registers are used as flags that can be raised but cannot be lowered back. The machine can move up and down between nodes and can access registers at the presently visited node and its ancestor nodes. This access is limited to using the registers as positive guards: it can verify that a flag is up, but not that it is down.

In general, an ID of an automaton is a triple $\langle\, q, T, w \,\rangle$, where q is a state, T is a tree of knowledge, and w is the currently visited node of T, called the *current apple*. Initially the tree consists of the single root node and all the registers at the root are zero (all flags are down). The computation starts in a fixed initial state q_0^0 with $w = \varepsilon$. The further processing is alternating: being in a universal state, the machine can split the computation into several branches (by selecting a number of successor states). However, in a universal step, both T and w remain unchanged; those can be modified in existential steps in the following ways:

1. the apple can go up to a father of w or down to a nondeterministically chosen child of w;

2. a selected flag can be raised at a given ancestor node of w;
3. the machine can check if a selected flag is up at a given ancestor node of w;
4. a new child of w can be added to the tree T.

In case 3, if the flag in question is down, the computation gets stuck; one cannot make negative tests. The same happens in case 1 if we attempt to move up when w is a root or to move down when w has no children. In case 2, raising a flag which is already 1 has no effect (and remains unnoticed). In all cases but 4, machine instructions also specify the next state. However, when a new node is created at level ℓ the automaton has no choice and must enter state q_0^ℓ, a designated initial state for that level.

The behaviour of Eden automata somehow resembles tree-walking automata [1], but there are essential differences. First of all, our automata build their own trees rather than recognise them. Also the node branching is potentially unbounded. We also notice that tree automata with additional memory were introduced by Kaminski and Tan [4].

Our main result is obtained in two steps. First, we show that the halting problem for Eden automata reduces to the decision problem for positive formulas. In fact, a LOGSPACE reduction from automata to formulae is also possible at the cost of a slight complication in the definition of automata. This way we obtain a machine model exactly equivalent in complexity to the decision problem for our formulas. But for the lack of space we do not pursue this issue here.

In the second step we prove the 2-EXPTIME hardness for automata, using a direct Turing Machine simulation. It is worth noting that the main difficulty here is to ensure the unequivocality of representation, namely that a single code represents a single entity, but not any other, which is the result of the situation that Eden automata are earthly entities with too limited memory to deal with the complexity of Eden citizens such as trees of knowledge. Putting together the two reductions we obtain the doubly-exponential lower bound for provability.

References

[1] A.V. Aho and J.D. Ullman. Translations on a context-free grammar. *Inf. Control*, 19(5):439–475, 1971.

[2] Gilles Dowek and Ying Jiang. Eigenvariables, bracketing and the decidability of positive minimal predicate logic. *Theor. Comput. Sci.*, 360(1–3):193–208, 2006.

[3] Gilles Dowek and Ying Jiang. Enumerating proofs of positive formulae. *Comput. J.*, 52(7):799–807, 2009.

[4] Michael Kaminski and Tony Tan. Tree automata over infinite alphabets. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science*, volume 4800 of *LNCS*, pages 386–423. Springer-Verlag, Berlin, Heidelberg, 2008.

[5] G.E. Mints. Solvability of the problem of deducibility in LJ for a class of formulas not containing negative occurrences of quantifiers. *Steklov Inst.*, 98:135–145, 1968.

[6] Ivar Rummelhoff. *Polymorphic* Π1 *Types and a Simple Approach to Propositions, Types and Sets.* PhD thesis, University of Oslo, 2007.

Unfolding Nested Patterns and Copatterns

Anton Setzer[1*]

Department of Computer Science, Swansea University, Singleton Park, Swansea SA2 8PP, UK
a.g.setzer@swan.ac.uk

Because of the importance of interactive programs, which result in potentially infinite computation traces, coalgebraic data types play an important rôle in computer science. Coalgebraic data types are often represented in functional programming as codata types. Implicit in the formulation of codata types is that every element of the coalgebra is introduced by a constructor. Our first result in this talk is to show that this assumption results in an undecidable equality.

In order to have a decidable equality modifications were added to the rules in Agda and Coq. This results in lack of subject reduction in the theorem prover Coq and a formulation of coalgebraic types in Agda, which is severely restricted.

In our joint article [1] we demonstrated how, when following the well known fact in category theory that final coalgebras are the dual of initial algebras, we obtain a formulation of final and weakly final coalgebras which is completely symmetrical to that of initial or weakly initial algebras. Introduction rules for algebras are given by the constructors, whereas elimination rules correspond to recursive pattern matching. Elimination rules for coalgebras are given by destructors, whereas introduction rules are given by recursive copattern matching. The resulting theory was shown to fulfil subject reduction. The article [1] allowed nested pattern and copattern matching and even mixing of the two. That article allows as well full recursion and therefore is not normalising.

In the second part of our talk we will investigate how to represent codata types which are often given by having several constructors, in this coalgebraic setting using a suitable abbreviation mechanism. Functions can be almost written in the same way as using codata types, while maintaining the fact that there are no special restrictions on the reductions as needed when using codata types.

In the third part, we will show how to reduce nested copattern and pattern matching to simple (non-nested) pattern matching. We will extend the algorithm replacing nested pattern matching for algebras in [2]. Then we introduce two versions of (co)recursion operators. One is allows full (co)recursion (and could be replaced by (co)case distinction and the Y-combinator), and the other corresponds to primitive (co)recursion, which is essentially $F_{(co)rec}$ in [3]. All terms can now be translated using the full (co)recursion operators into combinatorial terms. Terms which allow the translation into primitive (co)recursion operators should be those which are to be passed by a termination checker in an implementation of the calculus in [1].

As an example the full and primitive (co)recursion operator for $\mathbb{N}$ and Stream are:

$$
\begin{array}{l}
\mathrm{P}_{\mathbb{N},A} : A \to (\mathbb{N} \to A \to A) \to \mathbb{N} \to A \\
\mathrm{P}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ 0 \quad = \quad \mathrm{step}_0 \\
\mathrm{P}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ (\mathrm{suc}\ n) \quad = \quad \mathrm{step}_{\mathrm{suc}}\ n\ (\mathrm{P}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ n)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{R}_{\mathbb{N},A} : ((\mathbb{N} \to A) \to A) \to ((\mathbb{N} \to A) \to \mathbb{N} \to A) \to \mathbb{N} \to A \\
\mathrm{R}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ 0 \quad = \quad \mathrm{step}_0\ (\mathrm{R}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}) \\
\mathrm{R}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ (\mathrm{suc}\ n) \quad = \quad \mathrm{step}_{\mathrm{suc}}\ (\mathrm{R}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}})\ n
\end{array}
$$

*Supported by EPSRC grant EP/G033374/1, theory and applications of induction-recursion. Part of this work was done while the second author was a visiting fellow of the Isaac Newton Institute for Mathematical Sciences, Cambridge, UK.

$$\begin{array}{l}\mathrm{coP}_{\mathrm{Stream},A} : (A \to \mathbb{N}) \to (A \to (\mathrm{Stream} + A)) \to A \to \mathrm{Stream}\\ \mathrm{head}\ (\mathrm{coP}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}}\ a) \quad = \quad \mathrm{step}_{\mathrm{head}}\ a\\ \mathrm{tail}\ \ (\mathrm{coP}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}}\ a) \quad = \quad \mathrm{case}_{\mathrm{Stream},A,\mathrm{Stream}}\ (\lambda\ s.s)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathrm{coP}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}})\ (\mathrm{step}_{\mathrm{tail}}\ a)\end{array}$$

$$\begin{array}{l}\mathrm{case}_{A,B,C} : (A \to C) \to (B \to C) \to (A + B) \to C\\ \mathrm{case}_{A,B,C}\ \mathrm{step}_{\mathrm{inl}}\ \mathrm{step}_{\mathrm{inr}}\ \ (\mathrm{inl}\ a) \quad = \quad \mathrm{step}_{\mathrm{inl}}\ a\\ \mathrm{case}_{A,B,C}\ \mathrm{step}_{\mathrm{inl}}\ \mathrm{step}_{\mathrm{inr}}\ \ (\mathrm{inr}\ b) \quad = \quad \mathrm{step}_{\mathrm{inr}}\ b\end{array}$$

$$\begin{array}{l}\mathrm{coR}_{\mathrm{Stream},A} : ((A \to \mathrm{Stream}) \to A \to \mathbb{N}) \to ((A \to \mathrm{Stream}) \to A \to \mathrm{Stream}) \to \mathrm{Stream}\\ \mathrm{head}\ (\mathrm{coR}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}}\ a) \quad = \quad \mathrm{step}_{\mathrm{head}}\ (\mathrm{coR}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}})\ a\\ \mathrm{tail}\ (\mathrm{coR}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}}\ a) \quad = \quad \mathrm{step}_{\mathrm{tail}}\ (\mathrm{coR}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}})\ a\end{array}$$

As a simple example consider for some fixed $N : \mathbb{N}$ the stream (cycle n), which is informally written as $(n, n-1, \cdots, 0, N, N-1, N-2, \cdots 0, N, N-1, \cdots)$:

$$\begin{array}{lcl}\mathrm{cycle} : \mathbb{N} \to \mathrm{Stream}\\ \mathrm{head}\ (\mathrm{cycle}\ n) & = & n\\ \mathrm{tail}\ \ (\mathrm{cycle}\ 0) & = & \mathrm{cycle}\ N\\ \mathrm{tail}\ \ (\mathrm{cycle}\ (\mathrm{suc}\ n)) & = & \mathrm{cycle}\ n\end{array}$$

The algorithm for replacing it by non-nested (co)pattern matching yields:

$$\begin{array}{lcl}\mathrm{cycle} : \mathbb{N} \to \mathrm{Stream}\\ \mathrm{head}\ (\mathrm{cycle}\ n) & = & n\\ \mathrm{tail}\ \ (\mathrm{cycle}\ n) & = & \mathrm{cycle}_0\ n\end{array} \qquad \begin{array}{lcl}\mathrm{cycle}_0 : \mathbb{N} \to \mathrm{Stream}\\ \mathrm{cycle}_0\ 0 & = & \mathrm{cycle}\ N\\ \mathrm{cycle}_0\ (\mathrm{suc}\ n) & = & \mathrm{cycle}\ n\end{array}$$

which in this case can be replaced by primitive (co)recursion:

$$\begin{array}{l}\mathrm{cycle} : \mathbb{N} \to \mathrm{Stream}\\ \mathrm{cycle} = \mathrm{coP}_{\mathrm{Stream},\mathbb{N}}\ (\lambda n.n)\ \mathrm{cycle}_1\end{array} \qquad \begin{array}{l}\mathrm{cycle}_1 : \mathbb{N} \to (\mathrm{Stream} + \mathbb{N})\\ \mathrm{cycle}_1 = \mathrm{P}_{\mathbb{N},(\mathrm{Stream}+\mathbb{N})}\ (\mathrm{inr}\ N)\ (\lambda n, x.\mathrm{inr}\ n)\end{array}$$

By Mendler [4] and Geuvers [3] it follows that the restriction to primitive (co)recursion operators is fully normalising, which implies that a termination checked version of the calculus in [1] is normalising.

We would like to thank the anonymous referees for valuable comments on earlier versions of this abstract.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 27–38, 2013.

[2] Chi Ming Chuang. *Extraction of Programs for Exact Real Number Computation using Agda*. PhD thesis, Dept. of Computer Science, Swansea University, Swansea SA2 8PP, UK, March 2011. Availabe from `http://www.cs.swan.ac.uk/~csetzer/articlesFromOthers/index.html`.

[3] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Informal proceedings of the 1992 workshop on Types for Proofs and Programs, Bastad 1992, Sweden*, pages 183 – 207, 1992.

[4] N. P. Mendler. Inductive types and type constraints in second-order lambda calculus. In *Proceedings of the Second Symposium of Logic in Computer Science. Ithaca, N.Y.*, pages 30 – 36. IEEE, 1987.

Universe Polymorphism and Inference in Coq

Matthieu Sozeau

INRIA Paris & PPS, Paris 7 University
matthieu.sozeau@inria.fr

Abstract

Type theories such as the Calculus of Inductive Constructions maintain a universe hierarchy to prevent logical paradoxes. To ensure consistency while not troubling the user with this necessary information, systems using typical ambiguity were designed, which handle universes implicitly. Universes are seen as floating and their usage determines a graph of constraints, which must statisfy a property of acyclicity to ensure consistency. While this is a useful mechanism, there are a number of situations where the user would like to have developments made *polymorphic* on the universes used and instantiate his constructions at different levels. Typically, this is necessary to nest different instances of a given structure containing universes. We present an elaboration from terms using typical ambiguity into explicit terms which also accomodates universe polymorphism, i.e. the ability to write a term once and use it at different universe levels. Elaboration relies on an enhanced type inference algorithm to provide the freedom of typical ambiguity while also supporting polymorphism, in a fashion similar to usual Hindley-Milner polymorphic type inference. This elaboration is implemented as a drop-in replacement for the existing universe system of Coq and has been benchmarked favorably against the previous version. We demonstrate how it provides a solution to a number of formalization issues present in the original system.

The Calculus of Inductive Constructions implemented in the Coq proof assistant relies on a system of universes to ensure logical consistency. It avoids paradoxes coming from the infamous $\mathsf{Type} : \mathsf{Type}$ rule of system U- [1] by stratifying the universes used in a development, giving them names ("levels") and building up a graph of their inclusion relationships. It ensures that these relationships are coherent at any given point of a development. This can be checked by showing that there is always an assignment of natural numbers to universe levels that satisfy the universe constraints, which implies that the stratified universe levels map to the well-founded suite of universes $\mathsf{Type}_0 < \mathsf{Type}_1 < .. < \mathsf{Type}_n$.

To avoid forcing the user to work directly with levels, a system of typical ambiguity is used in Coq, where one can leave out the names of universes involved in a definition and let the system generate fresh universe variables and associated constraints. However in its current form this system has a serious limitation as it does not allow one to make definitions that can be instantiated at different levels: levels are always global. Let's look at a simple example. The polymorphic identity function id is written:

$$\texttt{Definition}\ \mathsf{id}\ (A : \mathsf{Type})\ (a : A) := a$$

It takes a type A in an unspecified universe, an object of that type and returns it. The actual CIC term constructed by this definition involves a fresh universe variable (say l) and its type-checking judgement has conclusion:

$$\vdash (\lambda(A : \mathsf{Type}_l)(a : A), a) : \Pi(A : \mathsf{Type}_l),\ A \to A$$

The rule for type-checking product types in CIC allows us to derive for the type of the identity function a judgment:

$$\vdash \Pi(A : \mathsf{Type}_l),\ A \to A : \mathsf{Type}_{l+1 \sqcup l}$$

Where $i \sqcup j$ represents the least upper bound of levels i and j. Here it can be simplified as Type_{l+1}. Suppose we want to apply the identity function to itself, we'll have to typecheck:

$$\vdash \mathsf{id}\ (\Pi(A : \mathsf{Type}_l),\ A \to A)\ \mathsf{id} : (\Pi(A : \mathsf{Type}_l),\ A \to A)$$

However, this will not be allowed, as typechecking of the application of id to the type $\Pi(A : \mathsf{Type}_l),\ A \to A$ involves a *universe constraint*, namely that the type of the argument, here Type_{l+1} be less or equal to id's first argument type, here Type_l. This would lead to an inconsistency and is hence rejected by the type checker. This happens because the system of universes fixed the level of id's first argument to a level l and it's usage is henceforth restricted. Applying id to a copy id' of itself would however work. Clearly, forcing the user to clone his definitions repeatedly (and accordingly, all proofs depending on them) to get them to typecheck at different levels is not a practical solution.

Instead of fixing id's level l at the global level, we can interpret the definition as being parametric on its level and make each of its occurrences generate a fresh one. I.e. we can consider $\mathsf{id}_l : \Pi(A : \mathsf{Type}_l),\ A \to A$ for any given l. It is now possible to instantiate id at different levels in a single definition:

$$\vdash \mathsf{id}_{l+1}\ (\Pi(A : \mathsf{Type}_l),\ A \to A)\ \mathsf{id}_l : (\Pi(A : \mathsf{Type}_l),\ A \to A)$$

This is the basic idea of the system of universe polymorphic definitions we are implementing in Coq. It is inspired by Harper & Pollack's design for LEGO [2], which we extended with a new universe elaboration algorithm that supports typical ambiguity and polymorphism. The system is an elaboration from definitions with implicit universes into terms decorated with universe information. It involves two changes to Coq's architecture:

1. A move from constraint generation to constraint *checking* in Coq's kernel as constraints are now given by the type inference algorithm. This somewhat simplifies the code of the trusted code base. The representation of terms is changed in a minimal way by adding universe instances to constants, inductive types and constructors.

2. An extension of the type inference algorithm that handles typical ambiguity and minimization of constraints associated to universe polymorphic definitions. It produces a term decorated with universe instances and a set of constraints to check.

The main originality of this work is in the design of the constraint simplification algorithm which decorates universe polymorphic definitions using a miminal set of universes. The system has been implemented and tested on examples such as the Homotopy Type Theory library (`http://github.com/HoTT/HoTT`) where it exhibited good performance.

Compared to Agda's implementation of universe polymorphism, it does not introduce "framework types" and deals with cumulativity and typical ambiguity.

References

[1] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

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

Fully abstract semantics of $\lambda\mu$ with explicit substitution in the π-calculus

Steffen van Bakel and Maria Grazia Vigliotti

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, UK

The research we want to present is part of an ongoing investigation into the suitability of classical logic in the context of programming languages with control. Rather than looking at how to encode known control features into calculi like the λ-calculus [5], $\lambda\mu$ [14], or $\Lambda\mu$ [10], as has been done in great detail by others, we focus on trying to understand what is exactly the notion of computation that is embedded in $\lambda\mu$ by mapping that calculus into the (perhaps) better understood π-calculus [13].

Over the last two decades, the π-calculus and its dialects have proven to give an interesting and expressive model of computation. Encodings of variants of the pure λ-calculus [7, 5] started with [13], which quickly led to more thorough investigations. For these encodings, over the years strong properties have been shown like soundness, completeness, termination, and full abstraction. The strength of these results has encouraged researchers to investigate the possibility of encodings into the π-calculus of various calculi that have their foundation in classical logic, as done in, for example, [12, 1, 8, 6]. From these papers it might seem that the encoding of such calculi comes at a great price; for example, to encode typed $\lambda\mu$, [12] needs to consider a version of the π-calculus that is not only strongly typed, but, moreover, allows reduction under guard and under replication; [1] shows preservation of reduction in $\mathcal{X}$ [2] only with respect to $\sqsubseteq_{\mathrm{c}}$, the contextual ordering; [8] defines a non-compositional encoding of $\overline{\lambda}\mu\tilde{\mu}$ [9] that strongly depends on recursion, and does not regard the logical aspect at all.

In [4] we set out to understand the contextual reduction rule of $\Lambda\mu$ better by mapping $\Lambda\mu$ into π, but following the novel approach of *output-based interpretations*, that we first studied in [3] for the λ-calculus; this approach was compared to the traditional one in [11]. We showed in [4] that by extending the output-based interpretation $\llbracket M \rrparenthesis a$ of [3] (where $M \in \Lambda$ and a the name given to its anonymous output), adding cases for μ-binding and naming, we get a very natural translation of terms to processes that shows that structural substitution is just a variant of application. We defined $\Lambda\mu\mathbf{x}$ by adding explicit logical and structural substitution, and defined the translation of $\Lambda\mu\mathbf{x}$ terms into the π-calculus by:

$$
\begin{array}{rcl@{\qquad}rcl}
\llbracket x \rrparenthesis a & \triangleq & x(u).\,!\,u \twoheadrightarrow a & \llbracket M\langle \beta := N\cdot\gamma\rangle \rrparenthesis a & \triangleq & (\nu\beta)\,(\llbracket M \rrparenthesis a \mid \llbracket \beta := N\cdot\gamma \rrparenthesis) \\
\llbracket \lambda x.M \rrparenthesis a & \triangleq & (\nu xb)\,(\llbracket M \rrparenthesis b \mid \overline{a}\langle x, b\rangle) & \llbracket \alpha := Q\cdot\gamma \rrparenthesis & \triangleq & !\,\alpha(v,d).\,(\llbracket v := Q \rrparenthesis \mid !\,d \twoheadrightarrow \gamma) \\
\llbracket MN \rrparenthesis a & \triangleq & (\nu c)\,(\llbracket M \rrparenthesis c \mid \llbracket c := N\cdot a \rrparenthesis) & \llbracket [\beta]M \rrparenthesis a & \triangleq & \llbracket M \rrparenthesis \beta \\
\llbracket x := N \rrparenthesis & \triangleq & !\,(\nu w)\,\overline{x}\langle w\rangle.\,\llbracket N \rrparenthesis w & \llbracket \mu\gamma.M \rrparenthesis a & \triangleq & (\nu b)\,(\llbracket M[a/\gamma] \rrparenthesis b) \\
\llbracket M\langle x := N\rangle \rrparenthesis a & \triangleq & (\nu x)\,(\llbracket M \rrparenthesis a \mid \llbracket x := N \rrparenthesis)
\end{array}
$$

Our translation fully respects the explicit head reduction $\rightarrow_{\mathbf{x}\mathrm{H}}$ (a notion of reduction with explicit substitution that replaces only the head variable in a term), modulo contextual equivalence, using renaming of output and garbage collection ($\rightarrow_\pi^* \sim_{\mathrm{C}}$); we have shown the following properties for $\llbracket \cdot \rrparenthesis \cdot$ in [4]:

(*Operational Soundness*) : $M \rightarrow_{\mathbf{x}\mathrm{H}} N \Rightarrow \llbracket M \rrparenthesis a \rightarrow_\pi^* \sim_{\mathrm{C}} \llbracket N \rrparenthesis a$;
(*Adequacy*) : $M =_\beta N \Rightarrow \llbracket M \rrparenthesis a \sim_{\mathrm{C}} \llbracket N \rrparenthesis a$;
(*Operational Completeness*) : $\llbracket M \rrparenthesis a \rightarrow_\pi P \Rightarrow \exists N\,[\,P \rightarrow_\pi^* \sim_{\mathrm{C}} \llbracket N \rrparenthesis a \;\&\; M \rightarrow_{\mathbf{x}}^* N\,]$;

(*Type preservation*) : $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta \Rightarrow \llbracket M \rrbracket a : \Gamma \vdash_{\pi} a{:}A, \Delta$;
(*Termination*) : $M{\downarrow} \Rightarrow \llbracket M \rrbracket a \downarrow_{\mathrm{L}\pi}$.

The details of our results stress that the π-calculus constitutes a very powerful abstract machine indeed: although the notion of structural reduction in $\lambda\mu$ is very different from normal β-reduction, no special measures had to be taken in order to be able to express it; the component of our encoding that deals with pure λ-terms is almost exactly that of [3] (ignoring for the moment that substitution is modelled using a guard, which affects also the interpretation of variables), but for the use of replication in the case for application. In fact, the distributive character of structural substitution is dealt with entirely by congruence.

After finishing [4], we proved full abstraction results, by showing $M \sim N \Leftrightarrow \llbracket M \rrbracket a \sim_{\mathrm{C}} \llbracket M \rrbracket a$ where $\sim$ is an equivalence between terms, and $\sim_{\mathrm{C}}$ is contextual equivalence. To achieve this, we had to restrict our interpretation to $\lambda\mu$ (note that $\llbracket \mu\alpha.[\beta]M \rrbracket a \triangleq \llbracket M[a/\alpha] \rrbracket \beta$). We needed to characterise what is exactly the equivalence between terms in $\lambda\mu$ that is representable in the π-calculus through $\llbracket \cdot \rrbracket \cdot$: this turns out to be *weak* equivalence, that equates terms that have the same Lévy-Longo trees, but of course defined for $\lambda\mu$. We defined a notion of equivalence $\sim_{\mathbf{x}\mathrm{H}}$ between terms of $\lambda\mu\mathbf{x}$ that equates also terms that have no weak head-normal form, and show that terms are equivalent with respect to $\sim_{\mathbf{x}\mathrm{H}}$ if and only if their images under $\llbracket \cdot \rrbracket \cdot$ are contextually equivalent. We then generalised this to equivalences generated by head reduction, normal reduction, and approximation, respectively, and show that all coincide: this will lead to our main result: $M \sim_{\beta\mu} N \Leftrightarrow \llbracket M \rrbracket a \sim_{\mathrm{C}} \llbracket N \rrbracket a$.

References

[1] S. van Bakel, L. Cardelli, and M.G. Vigliotti. From $\mathcal{X}$ to π; Representing the Classical Sequent Calculus in the π-calculus. In *CL&C'08*, Reykjavik, 2008.

[2] S. van Bakel and P. Lescanne. Computation with Classical Sequents. *Mathematical Structures in Computer Science*, 18:555–609, 2008.

[3] S. van Bakel and M.G. Vigliotti. A logical interpretation of the λ-calculus into the π-calculus, preserving spine reduction and types. In *CONCUR'09*, LNCS 5710, pp. 84 – 98. Springer, 2009.

[4] S. van Bakel and M.G. Vigliotti. An Output-Based Semantics of $\Lambda\mu$ with Explicit Substitution in the π-calculus - Extended Abstract. In *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference*, LNCS 7604, pp. 372–387. Springer, 2012.

[5] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1984.

[6] E. Beffara and V. Mogbil. Proofs as Executions. In *Proceedings of TCS 2012 - 7th IFIP TC 1/WG 2.2 International Conference*, LNCS 7604, pp. 280–294. Springer, 2012.

[7] A. Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

[8] M. Cimini, C. Sacerdoti Coen, and D. Sangiorgi. $\overline{\lambda}\mu\tilde{\mu}$ calculus, π-calculus, and abstract machines. In *EXPRESS'09*, 2009.

[9] P.-L. Curien and H. Herbelin. The Duality of Computation. In *ICFP'00*, volume 35.9 of *ACM Sigplan Notices*, pp. 233–243. ACM, 2000.

[10] Ph. de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In *LPAR'94*, LNCS 822, pp. 31–43. Springer, 1994.

[11] D. Hirschkoff, J.-M. Madiot, and D. Sangiorgi. Duality and i/o-Types in the π-Calculus. In *CONCUR 2012*, LNCS 7454, pp. 302–316. Springer, 2012.

[12] K. Honda, N. Yoshida, and M. Berger. Control in the π-Calculus. In *Fourth ACM-SIGPLAN Continuation Workshop* (CW'04), 2004.

[13] R. Milner. Functions as processes. *Math. Structures in Computer Science*, 2(2):269–310, 1992.

[14] M. Parigot. An algorithmic interpretation of classical natural deduction. In *LPAR'92*, LNCS 624, pp. 190–201. Springer, 1992.

A very generic implementation of data-types with binders in Coq

Benjamin Werner

Ecole Polytechnique

Abstract

Formalizing structures with binders, like first-order logic, lambda-calculus or programming languages in a system like Coq is a very common problem. The recent years have seen a lot of work devoted to it, which now gives us a quite clear picture of the respective advantages and drawbacks of various approaches: De Bruijn indices, nominal approach, the locally nameless approach which compromises between the former two, and many other variants. In any case, when starting a formalization, the user has to make an early choice about what encoding will be used. Furthermore, every approach still comes with a boilerplate part of work (typically defining the lifting functions for de Bruijn indices or stating and proving the co-finite induction principles for the locally nameless approach).

We propose a generic data-type which allows to encode all languages with binders, provided all operators have a fixed arity. This data-type comes with a full set of functions and induction principles, thus allowing the user to avoid boilerplate work, and also to switch between diverse approaches in the same development.

1 Motivation and setting

When defining the syntax of a specific language in type theory, the obvious way is to use an inductive type: each operator of the language is mapped to a corresponding constructor of the inductive type; both having the same fixed arity. This approach generalized to many-sorted languages by using mutual inductive types.

Things are more complicated when the formalized language yields binders. It is notorious one has to make an almost religious choice between various solutions :

- named variables, which involves dealing with explicit α-conversion and various tricky points about freshness of identifiers,
- "full" De Bruijn indices, which involves defining various lifting operators, which then appear in further definitions,
- locally nameless approach, where De Bruijn indices are restricted to variables bound in the term,
- trying to mimic at least the spirit of the Higher-Order Abstract Syntax approach, in which the binder of the meta formalism is used, as it can be done in weaker formalisms like LF.

In all cases where De Bruijn indices are involved, one has to build a specific induction principle. This is particularly the case in the locally nameless approach, where Aydemir et. al have designed a clever co-finite quantification induction scheme.

In all cases, there is thus a non-negligible amount of boilerplate work involved. The aim of this work is to provide a package which allows the user to handle, in a smooth way, all the approaches but the one using only named variables. We do this by proposing a generic type which :

- Allows the encoding of any language with binders,
- comes with generic lifting functions,
- allows the automatic generation of induction schemes for a given language, including the co-finite quantification scheme for the locally nameless approach and the general scheme for traditional De Bruijn approach.

2 The construction

We use the SSreflect approach. Given a type for the sorts of the language, over which equality is decidable (an `eqtype` in SSR terminology) we define :

```
Inductive open :=
 | Db : nat -> open
 | Var : nat * sort -> open
 | Op : op -> vector -> open
with
 vector :=
 | vn
 | vc : open -> vector -> vector.
```

Here `open` stand for open terms. We see that the arity of operator is not yet fixed, and also that, for now, the operators are coded by natural numbers.

A given language is then defined by a mapping of each operator to an arity :

```
ar : nat -> option (((seq (sort * (seq sort))) * sort)%type).
```

Where, for instance, the arity $Some([s_1, [t_1; t_2]; (s_2, [])], s)$ should be understood as : an operator which builds a term of sort s, provided it is given two arguments :

- a second one of sort s_2, in which no variable is bound,
- a first one on sort s_1, in which two variables are bound, of respective sorts t_1 and t_2.

We the provide a notion of being well-sorted for a term, as well as lifting functions, replacing De Bruijn indices by named variables and the other way around, etc. The definition for well-sorted follows a typical SSReflect style approach, being defined as a recursive function rather than an inductive predicate :

```
sortof (t:open)(c:seq sort) : option sort
```

The main work is to build the various flavors of induction schemes automatically from the description of the operator arities. Currently we provide one induction schemes for "open" terms, which take into account the well-sortedness constraint, as well as a first version for a locally nameless induction scheme.

One technical difficulty is that all construction have to be mutually recursive and dual for terms and term vectors.

To be fully comfortable, this package will need some clever syntactic sugaring. For now, we provide some sketchy studies of typed λ-calculi.

References

Engineering Formal Metatheory, by Aydemir, Charguéraud, Pierce, Pollack and Weirich. Proceedings of POPL' 08. IEEE, 2008.

Author Index