TYPES 2014

Types for Proofs and Programs

20th Meeting

Paris, France

12-15 May, 2014

Book of Abstracts

Preface

This is the collection of the abstracts of the 20th Conference “Types for Proofs and Programs”, TYPES 2014, to take place in Paris, France, 12–15 May 2014. Paris was chosen this year to take advantage of the synergy created by the special trimester on *Semantics of proofs and certified mathematics* at the Institut Henri Poincaré, 22 April - 11 July 2014.

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 2014 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 programme committee:

- Andreas Abel, Chalmers University of Technology and Gothenburg University, Sweden
- Andrej Bauer, Fakulteta za matematiko in fiziko, Ljubljana, Slovenia
- Małgorzata Biernacka, University of Wroclaw, Poland
- Lars Birkedal, Aarhus University, Denmark
- Paul Blain Levy, University of Birmingham, UK
- Herman Geuvers, Radboud University and Eindhoven University of Technology, Netherlands
- Hugo Herbelin, INRIA Paris-Rocquencourt, France (co-chair)
- Pierre Letouzey, University Paris-Diderot, France (co-chair)
- Ralph Matthes, IRIT, CNRS and University of Toulouse, France
- Conor McBride, University of Strathclyde, UK
- Luís Pinto, University of Minho, Braga, Portugal
- Claudio Sacerdoti, University of Bologna, Italy
- Aleksy Schubert, University of Warsaw, Poland
- Matthieu Sozeau, INRIA Paris-Rocquencourt, France (co-chair)
- Thomas Streicher, TU Darmstadt, Germany

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

- Thierry Coquand, Chalmers University of Technology and Gothenburg University, Sweden
- Xavier Leroy, INRIA Paris-Rocquencourt, France
- Andy Pitts, Cambridge University, UK

as well as 39 contributed talks.

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 the invitation. And, of course, the effort of the programme committee members is gratefully acknowledged.

Institut Henri Poincaré kindly offered to have the conference hosted in its building, rue Pierre et Marie Curie, Paris 5th. Inria Paris-Rocquencourt provided general support for the organization, with special thanks to Chantal Girodon and Lindsay Polienor. Poster was conceived by Bastien Sozeau. Reviewing was made on Easychair.

April 23, 2014
Hugo Herbelin, Pierre Letouzey, Matthieu Sozeau

Invited Talks

A cubical set model of type theory

Thierry Coquand

Computer Science and Engineering Department
University of Gothenburg
Sweden

We present a possible constructive justification of the axiom of univalence. Roughly speaking, the computations for dependent type theory can be described with lambda terms (extended with constructors and constants for primitive recursive functions), while for dependent type theory with the axiom of univalence, computations are described using a nominal extension of lambda calculus (with some additional "face" operations). Constants corresponding to the elimination rule for equality can then be described by induction on the types. We describe a possible implementation corresponding to this semantics. This also provides a model of types such as the circle, or the operation of propositional truncation. In particular, we get a computational justification of the axiom of description.

Formal verification of a static analyzer: abstract interpretation in type theory

Xavier Leroy

Inria Paris-Rocquencourt
`xavier.leroy@inria.fr`

(Joint work with David Pichardie, Sandrine Blazy, Jacques-Henri Jourdan, and Vincent Laporte.)

Abstract

Static analysis is the automatic inference and checking of simple properties of all executions of a program. Initially developed in the context of compilers to support code optimization, static analysis is very successful today for the formal verification of safety properties of critical software, owing to its good scalability. As is the case for all tools involved in the production and verification of critical software (compilers, code generators, program provers, model checkers), confidence in the results of a static analysis tool requires evidence that the tool is sound and correctly over-approximates all possible executions of the program. Such evidence can take the form of a soundness proof mechanized using a proof assistant [5, 4].

Abstract interpretation [2] is an elegant, powerful mathematical framework to define and reason about static analyses. In particular, it is not limited to so-called "non-relational" analyses (inferring properties of a single value or variable) and works naturally for "relational" analyses (inferring relations between several variables, such as linear inequalities). The classic presentation of abstract interpretation involves Galois connections. It has the advantage that, once the meaning of abstract data is chosen via a Galois connection, the abstract operators used by the static analyzer can, in principle, be derived systematically from the concrete semantics, in a way that is not only sound by construction, but also relatively optimal.

However, the theory of Galois connections is resolutely set-theoretical, involving non-computable functions and equational reasoning over set comprehensions, making it very hard to express in type theory and to use in a proof assistant such as Coq. To overcome this difficulty, Pichardie *et al* [6, 1] developed and mechanized an alternative presentation of abstract interpretation, using only the "γ" (concretization) part of Galois connections, viewed as relations "*concrete-datum* $\in$ *abstract-datum*". The calculational style is lost, and relative optimality is no longer guaranteed, but soundness proofs are easily conducted with a proof assistant.

In the context of the Verasco project, we are currently trying to scale Pichardie's approach all the way to the development and Coq verification of a realistic static analyzer based on abstract interpretation for the CompCert subset of the C language. Proper modular decomposition is crucial to build the appropriate abstractions as a hierarchy combining numerical and memory abstract domains. While the general interface of a non-relational domain is well known, giving such an interface for relational domains is more challenging, and so is formulating generic composition operators (such as reduced products) between such domains. Another enabling technique is the opportunistic use of validation a posteriori to obviate the need to prove complicated algorithms such as fixpoint iteration with widening and narrowing, or operations over polyhedra for relational domains of linear inequalities [3].

Acknowledgments This work is supported by the Verasco project (ANR-11-INSE-003) of Agence Nationale de la Recherche (ANR).

References

[1] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

[3] Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *Static Analysis - 20th International Symposium (SAS 2013)*, volume 7935 of *Lecture Notes in Computer Science*, pages 345–365. Springer, 2013.

[4] Paolo Herms, Claude Marché, and Benjamin Monate. A certified multi-prover verification condition generator. In *Verified Software: Theories, Tools, Experiments (VSTTE 2012)*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2012.

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

[6] David Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés.* PhD thesis, Université Rennes 1, 2005.

Nominal Sets and Dependent Type Theory

Andrew M. Pitts

Computer Laboratory
University of Cambridge
Cambridge CB3 0FD, UK

Nominal sets [3, 7] provide a mathematical theory of structures involving names and binding constructs, based on some simple, but subtle ideas going back to Fraenkel and Mostowski's symmetric models of set theory with atoms. The theory has been applied to programming language semantics, machine-assisted theorem proving and the design of functional and logical metaprogramming languages. In this talk I want to explore the relationship between nominal sets and dependent type theory, with the following two motivations in mind, both of which involve the nominal sets notion of *name abstraction*.

Homotopy Type Theory. The cubical sets model of homotopy type theory was introduced by Bezem, Coquand and Huber [1] using a category of presheaves. This category is equivalent to a category of nominal sets equipped operations for substituting contants 0 and 1 for names (the names in this case being names of cartesian axes $x, y, z, \ldots$); see [6]. In the nominal version of the model, proofs of identity are given by name abstractions: abstracting a named direction x in an element a gives a path (proof of equality) from $a[0/x]$ to $a[1/x]$. In order to interpret dependent types, the category of nominal sets can be extended to a category with families [2, 4] in a straightforward way.

Constructive nominal logic. FreshML [8] adds name abstraction types to ML [5], allowing the user to declare inductively defined data involving name binding operations and define functions on such data using patterns involving bound names. The semantics of FreshML guarantees that programmers cannot break α-conversion, while allowing them to use a style close to informal practice when manipulating structures with bound names. I would very much like to have a similarly usable language that completes the following proportion:

$$\frac{\text{Agda}}{\text{Haskell}} = \frac{?}{\text{FreshML}}$$

Achieving this convincingly requires versions of the nominal sets notions of *freshness*, *name abstraction* and *name restriction* within constructive type theory that have good meta-theoretic properties and yet are syntactically simple from a user's point of view.

References

[1] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. Preprint, September 2013.

[2] Peter Dybjer. Internal type theory. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer Berlin Heidelberg, 1996.

[3] M. J. Gabbay. Foundations of nominal techniques: Logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic*, 17(2):161–229, 2011.

[4] M. Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, 1997.

[5] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[6] A. M. Pitts. An equivalent presentation of the Bezem-Coquand-Huber category of cubical sets. Preprint arXiv:1401.7807 [cs.LO], December 2013.

[7] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2013.

[8] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

Contributed Talks

Coinduction in Agda
Using Copatterns and Sized Types

Andreas Abel

Department of Computer Science and Engineering
Chalmers and Gothenburg University, Gothenburg, Sweden
andreas.abel@gu.se

Inductive data such as lists and trees is modeled category-theoretically as *algebra* where *construction* is the primary concept and elimination is obtained by initiality. In a more practical setting, functions are programmed by *pattern matching* on inductive data. Dually, coinductive structures such as streams and processes are modeled as *coalgebras* where *destruction* (or transition) is primary and construction rests on finality [Hag87]. Due to the coincidence of least and greatest fixed-point types [SP82] in lazy languages such as Haskell, the distinction between inductive and coinductive types is blurred in partial functional programming. As a consequence, coinductive structures are treated just as infinitely deep (or, non-well-founded) trees, and pattern matching on coinductive data is the dominant programming style. In total functional programming, which is underlying the dependently-typed proof assistants Coq [INR12] and Agda [Nor07], the distinction between induction and coinduction is vital for the soundness, and pattern matching on coinductive data leads to the loss of subject reduction [Gim96]. Further, in terms of expressive power, the *productivity checker* for definitions by coinduction lacks behind the termination checker for inductively defined functions.

It is thus worth considering the alternative picture that a *coalgebraic approach* to coinductive structures might offer for total and, especially, for dependently-typed programming. Understanding "algebraic programming" as defining functions by pattern matching, the dualization "coalgebraic programming" leads us to the notion of *copattern* matching. While patterns match the introduction forms of finite data, copatterns match on elimination contexts for infinite objects, which are applications (eliminating functions) and destructors/projections (eliminating coalgebraic types = Hagino's codatatypes). An infinite object such as a function or a stream can be defined by its behavior in all possible contexts. Thus, if we consider a set of copatterns covering all possible elimination contexts, plus the object's response for each of the copatterns, that object is defined uniquely. More concretely, a stream is determined by its head and its tail, thus, we can introduce a new stream object by giving two equations; one that specifies the value it produces if its head is demanded, and one for the case that the tail is demanded.

```
record Stream {i : Size} (A : Set) : Set where
  coinductive
  constructor _::_
  field head  : A
        tail  : ∀{j : Size< i} → Stream {j} A
open Stream public

zipWith : ∀{i A B C} (f : A → B → C) → Stream {i} A → Stream {i} B → Stream {i} C
head  (zipWith f s t) = f (head s) (head t)
tail  (zipWith f s t) = zipWith f (tail s) (tail t)
```

Another covering set of copatterns consists of head, head of tail and tail of tail. For instance, the stream of Fibonacci numbers can be given by the three equations, using a function zipWith $f\ s\ t$ which pointwise applies the binary function f to the elements of streams s and t.

```
fib : ∀{i} → Stream {i} ℕ
(       (head  fib)) = 0
(head  (tail   fib)) = 1
(tail  (tail   fib)) = zipWith _+_ fib (tail fib)
```

Taking the above equations as left-to-right rewrite rules, we obtain a strongly normalizing system. This is in contrast to the conventional definition of fib in terms of the stream constructor $h :: t$ by

$$\mathsf{fib} = 0 :: 1 :: \mathsf{zipWith}\ \_{+}\_\ \mathsf{fib}\ (\mathsf{tail}\,\mathsf{fib})$$

which, even if unfolded under destructors only, admits an infinite reduction sequence starting with tail fib $\longrightarrow 1 :: \mathsf{zipWith}\ \_{+}\_\ \mathsf{fib}\ (\mathsf{tail}\,\mathsf{fib}) \longrightarrow 1 :: \mathsf{zipWith}\ \_{+}\_\ \mathsf{fib}\ (1 :: \mathsf{zipWith}\ \_{+}\_\ \mathsf{fib}\ (\mathsf{tail}\,\mathsf{fib})) \longrightarrow \dots$ The crucial difference is that tail fib does not reduce if we choose the definition by copatterns above, since the elimination tail is not matched by any of the copatterns; only in contexts head or head of tail or tail of tail it is that fib springs into action.

Using definitions by copattern matching, we reduce productivity to termination and productivity checking to termination checking. As termination of a function is usually proven by a measure on the size of the function arguments, we prove productivity by well-founded induction on the size of the elimination context. For instance, fib is productive because the recursive calls occur in smaller contexts: at least one tail-destructor is "consumed" and, equally important, zipWith does not add any more destructors. The number of eliminations (as well as the size of arguments) can be tracked by sized types [HPS96], reducing productivity (and termination) checking to type checking. For a polymorphic lambda-calculus with inductive and coinductive types and patterns and copatterns, this has been spelled out in joint work with Brigitte Pientka [AP13]. An introductory study of copatterns and covering sets thereof can be found in previous work [APTS13].

A similar abstract has appeared under the title *Productive Infinite Objects via Copatterns* in the informal proceedings of NWPT 2013 (Nordic Workshop of Programming Theory, Tallinn, Estonia, November 2013), and under the title *Programming and Reasoning with Infinite Structures Using Copatterns and Sized Types* in the proceedings of ATPS 2014 (Arbeitstagung Programmiersprachen, Kiel, Germany, February 2014).

References

[AP13] Andreas Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. pages 185–196. ACM Press, 2013.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proc. of the 40th ACM Symp. on Principles of Programming Languages, POPL 2013*, pages 27–38. ACM Press, 2013.

[Gim96] Eduardo Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996. Thèse d'université.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proc. of the 23rd ACM Symp. on Principles of Programming Languages, POPL'96*, pages 410–423, 1996.

[INR12] INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.4 edition, 2012.

[Nor07] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept of Comput. Sci. and Engrg., Chalmers, Göteborg, Sweden, 2007.

[SP82] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

Coalgebraic Update Lenses

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

O'Connor [6] made the simple but very useful observation with deep consequences that the (very well-behaved) lenses à la Foster et al. [3] are nothing but coalgebras of the array comonads of Power and Shkaravska [7].

The put operation in these lenses is quite rigid in that a whole new view is merged into the source, there is no flexibility for speaking about small changes to the view. We advocate a generalization that is as simple as O'Connor's, but offers also this flexibility. The idea is to introduce updates (or changes, deltas, edits) that can be composed and applied to views. The generalization derives from the work on directed containers of Ahman et al. [1].

A lens in our generalized sense—an update lens—is parameterized by a fixed set S (of views), a monoid $(P, \mathsf{o}, \oplus)$ (of updates) and an action $\downarrow$ of the monoid on the set (describing the outcome of applying any given update on any given view).

These data, sometimes collectively called an act, define a comonad (D, ε, δ) by $D\,X = S \times (P \to X)$.

We define an update lens to be a coalgebra of this comonad. This is the same as having a set X and maps $\mathsf{lkp} : X \to S$ and $\mathsf{upd} : X \times P \to X$ satisfying the conditions

$$\begin{aligned} \mathsf{upd}\,(x, \mathsf{o}) &= x \\ \mathsf{upd}\,(\mathsf{upd}\,(x, p), p') &= \mathsf{upd}\,(x, p \oplus p') \\ \mathsf{lkp}\,(\mathsf{upd}\,(x, p)) &= \mathsf{lkp}\,x \downarrow p \end{aligned}$$

To have an update lens turns out to be equivalent to having a functor R from $\langle\!\langle S, (P, \mathsf{o}, \oplus) \downarrow\rangle\!\rangle$ to **Set**. Here $\langle\!\langle S, (P, \mathsf{o}, \oplus) \downarrow\rangle\!\rangle$ is the category where an object is an element of S, a map between $s, s' : S$ is an element of p such that $s \downarrow p = s'$, the identity on an object s is o and the composition of two maps p, p' is $p \oplus p'$.

An act S, $(P, \mathsf{o}, \oplus)$, $\downarrow$ also defines a monad (T, η, μ) by $T\,X = S \to P \times X$ (a compatible combination of the reader monad for S and the writer monad for $(P, \mathsf{o}, \oplus)$) that we have elsewhere [2] called the update monad. The algebras of this monad and update lenses model resp. comodel the same Lawvere theory.

Ordinary lenses for S are canonically related to update lenses for the act $(S, (P, \mathsf{o}, \oplus), \downarrow)$ where $(P, \mathsf{o}, \oplus)$ is the free monoid on the "overwrite" semigroup structure on S.

The algebraic treatment of ordinary lenses by Johnson et al. [5], compared to O'Connor's coalgebraic account by Gibbons and Johnson [4], extends to update lenses. The action $\downarrow$ defines a lifting of the writer monad for $(P, \mathsf{o}, \oplus)$ to category $\mathbf{Set}/S$. An update lens is essentially the same as an algebra of this lifted monad.

Acknowledgements Tarmo Uustalu thanks Andreas Abel for reminding him of O'Connor's work. This ongoing work is being supported by the University of Edinburgh Principal's Career Development PhD Scholarship, the ERDF funded Estonian CoE project EXCS and ICT National Programme project Coinduction, and the Estonian Research Council target-financed research theme 0140007s12 and grant no. 9475.

References

[1] D. Ahman, J. Chapman, T. Uustalu. When is a container a comonad? *Log. Methods in Comput. Sci.*, to appear. Conference version 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] D. Ahman, T. Uustalu. Update monads: cointerpreting directed containers. In R. Matthes, A. Schubert, eds., *Proc. of 19th Conf. on Types for Proofs and Programs (Toulouse, Apr. 2014)*, *Leibniz Proc. in Informatics*, Schloss Dagstuhl, to appear.

[3] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt. Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. *ACM Trans. on Program. Lang. and Syst.*, v. 29, n. 3, article 17, 2007.

[4] J. Gibbons, M. Johnson. Relating algebraic and coalgebraic descriptions of lenses. In F. Hermann, J. Voigtländer, eds., *Proc. of 1st Int. Wksh. on Bidirectional Transformations, BX 2012 (Tallinn, March 2012)*, v. 49 of *Electron. Commun. of EASST*, 16 pp, 2012.

[5] M. Johnson, R. Rosebrugh, R. J. Wood. Algebras and update strategies. *J. of Univ. Comput. Sci*, v. 16, n. 5, pp. 729748, 2010.

[6] R. O'Connor. Functor is to lens as applicative is to biplate: introducing multiplate. arXiv:1103.2841, 2011. (Paper presented at 2011 ACM SIGPLAN Wksh. on Generic Programming, Tokyo, Sept. 2011.)

[7] J. Power, O. Shkaravska. From comodels to coalgebras: state and arrays. In J. Adámek, S. Milius, eds., *Proc. of Wksh. on Coalgebraic Methods in Computer Science (Barcelona, March 2004)*), v. 106 of *Electr. Notes in Theor. Comput. Sci.*, pp. 297-314, Elsevier, 2004.

Coinitial semantics for redecoration of triangular matrices

Benedikt Ahrens and Régis Spadotti

Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier, Toulouse

In Martin-Löf type theory, simple inductive types—W-types—are characterized categorically as initial algebras of a polynomial functor. Dually, *co*inductive types are characterized as terminal *co*algebras of polynomial functors. In the case of coinductive types, the meta-theoretic notion of equality given by Martin-Löf's identity type is too weak: instead, the idea of *bisimilarity as equality* for coinductive data types was coined by Aczel [1].

The characterization of inductive types as initial algebras has been extended to some *heterogeneous*—also called *nested*—inductive data types, e.g., the type of λ-terms, in various formulations [4, 5]. The main goal of those works is to characterize not only the data type via a universal property, but rather the data type *equipped with a well-behaved substitution operation.*

In the present work we study a specific *co*inductive *heterogeneous* data type—the type family Tri of infinite triangular matrices—and its *redecoration* operation: the codata type is parametrized by a fixed type E for entries not on the diagonal, and indexed by another, *variable*, type A for entries on the diagonal. The respective types of its specifying destructors top and rest are given in Figure 1, together with the destructors for the coinductively defined bisimilarity relation on it. Equipped with the redecoration operation, the type Tri is shown by Matthes and Picard [6] to constitute what they call a "weak constructive comonad".

$$\frac{t : \mathsf{Tri}(A)}{\mathsf{top}_A(t) : A} \qquad \frac{t : \mathsf{Tri}(A)}{\mathsf{rest}_A(t) : \mathsf{Tri}(E \times A)}$$

$$\frac{t \sim t'}{\mathsf{top}(t) = \mathsf{top}(t')} \qquad \frac{t \sim t'}{\mathsf{rest}(t) \sim \mathsf{rest}(t')}$$

Figure 1: Destructors and bisimilarity for the coinductive family of setoids Tri

In this work, we first identify those weak constructive comonads as an instance of the more general notion of *relative comonad*, the dual to relative monads as introduced in [3]. Indeed, a weak constructive comonad is precisely a comonad relative to the functor $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$, the left adjoint to the forgetful functor.

Afterwards, we characterize the codata type Tri, equipped with the cosubstitution operation of redecoration, as a terminal object of some category. For this, we dualize the approach by Hirschowitz and Maggesi [5], who characterize the heterogeneous inductive type of lambda terms—equipped with a suitable substitution operation—as an initial object in a category of algebras for the signature of lambda terms. In their work, the crucial notions are the notion of monad and, more importantly, *module over a monad.* It turns out that more work than a simple dualization is necessary for two reasons:

- the lambda calculus can be seen as a monad on types and thus, in particular, as an endofunctor. The codata type Tri, however, associates to any *type* of potential diagonal elements a *setoid* of triangular matrices. We thus need a notion of comonad whose underlying functor is not necessarily endo: the already mentioned *relative* comonads;

- the category-theoretic analysis of the destructor rest is more complicated than that of the heterogeneous constructor of abstraction of the lambda calculus.

Finding a suitable categorical notion to capture the destructor rest and, more importantly, its interplay with the comonadic redecoration operation on Tri, constitutes the main contribution of the present work. These rather technical details shall not be given in this abstract, but are explained in a preprint [2].

Once we have found such a categorical notion, we can use it to give a definition of a “coalgebra” for the signature of infinite triangular matrices, together with a suitable notion of *morphism* of such coalgebras. We thus obtain a category of coalgebras for that signature. Any object of this category comes with a comonad relative to the aforementioned functor eq : Type $\to$ Setoid and a suitable comodule over this comonad, modeling in some sense the destructor rest. Our main result then states that this category has a terminal object built from the codata type Tri and its destructor rest, which are seen as a relative comonad and a comodule over that relative comonad, respectively. This universal property of coinitiality characterizes not only the codata type of infinite triangular matrices but also the bisimilarity relation on it as well as the redecoration operation.

All our definitions, examples, and lemmas have been implemented in the proof assistant Coq. The Coq source files and HTML documentation are available on the project web page [2].

We thank the anonymous referees for their helpful comments on this abstract.

References

[1] Peter Aczel. *Non-Well-Founded Sets*, volume 14 of *CSLI Lecture Notes*. Center for the Study of Languages and Information, 1988.

[2] Benedikt Ahrens and Régis Spadotti. Coinitial semantics for redecoration of triangular matrices. `http://benediktahrens.github.io/coinductives/`.

[3] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. In C.-H. Luke Ong, editor, *FOSSACS*, volume 6014 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2010.

[4] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS ’99, pages 193–202, Washington, DC, USA, 1999. IEEE Computer Society.

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

[6] Ralph Matthes and Celia Picard. Verification of redecoration for infinite triangular matrices using coinduction. In Nils Anders Danielsson and Bengt Nordström, editors, *TYPES*, volume 19 of *LIPIcs*, pages 55–69. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.

A Type Theory with Partial Equivalence Relations as Types

Abhishek Anand, Mark Bickford, Robert L. Constable, and Vincent Rahli

Cornell University

Abstract

A small core type language with intersection types in which a partial equivalence relation on closed terms is a type is enough to build the non-inductive types of Nuprl, including the types of dependent functions and partial functions. Using induction on natural numbers and intersection types, we build coinductive types; and using partial functions and coinductive types we build algebraic datatypes.

Introduction. Nuprl [6, 2] is a functional programming language based on a constructive dependent type theory with partial types called CTT. As in similar systems such as Coq [4] and Agda [5], it has dependent functions, inductive types, and a cumulative hierarchy of universes. In addition, CTT has dependent products, disjoint union, integer,[1], equality, set (or refinement) and quotient types [6]; intersection and union types [10]; image types [11]; computational approximation and equivalence types [12]; and is one of the only type theories with partial types [7, 8].

Allen gave a semantics of CTT where a type is a Partial Equivalence Relation (PER) on closed terms [1], which is connected to Russell's original definition of a type as "the range of significance of a propositional function." By allowing the theory to directly represent PERs as types, we can reformulate CTT using a smaller core of primitive type constructors. For example, the dependent function type can now be defined. Allen [1, pp.15] suggested such a type that represents PERs by combining the set and quotient types.

The `per` type constructor can turn PERs into types. Therefore, we need some primitives to express such PERs: `Base` is the type of closed terms (PERs are relations on closed terms) whose equality $\sim$ is Howe's computational equivalence [9]; equality (or identity) types to refer to already defined PERs[2]; our main logical operator is the intersection type constructor which is a uniform universal quantifier; the computational approximation type constructor $\preceq$ allows us to build PERs by imposing restrictions on their domains in terms of how terms compute.

When the partial, union and image types were added to Nuprl in the past we had to update the metatheory accordingly. Using the `per` constructor we can now add new types to Nuprl without changing the metatheory. We are already using this type in Nuprl and have defined several formerly primitive types using it, such as the quotient and partial types.

Nuprl's syntax. Nuprl is defined on top of an applied lazy untyped λ-calculus. We define the subset of this language that is of interest to us in this paper as follows:

$$\begin{aligned} A,B,R &::= t_1 \preceq t_2 \mid \texttt{Base} \mid \mathbb{U}_i \mid \texttt{per}(R) \mid \cap\, x{:}A.B[x] \mid t_1 = t_2 \in A \\ v &::= A \mid \underline{i} \mid \lambda x.t \mid \langle t_1, t_2\rangle \mid \texttt{Ax} \mid \texttt{inl}(t) \mid \texttt{inr}(t) \\ t &::= x \mid v \mid t_1\ t_2 \mid \texttt{fix}(t) \mid \texttt{let}\ x,y = t_1\ \texttt{in}\ t_2 \mid \texttt{let}\ x := t_1\ \texttt{in}\ t_2 \\ &\mid \texttt{if}\ t_1 < t_2\ \texttt{then}\ t_3\ \texttt{else}\ t_4 \mid \texttt{isint}(t_1,t_2,t_3) \mid \texttt{isaxiom}(t_1,t_2,t_3) \end{aligned}$$

where A, B, and R stand for types, $\underline{i}$ for an integer, v for a value, x for a variable, and t for a term. `Ax` is the unique canonical inhabitant of true propositions that do not have any nontrivial computational meaning in CTT, such as $0 = 0 \in \mathbb{N}$. The canonical form tests such as

[1]For efficiency issues, the integer type is a primitive type in Nuprl.

[2]We extended the definition of equality types so that the equality in T is not only a relation on T but also a relation on `Base` [3, Sec. 4.2.1].

$\texttt{isaxiom}$ allow us to distinguish between the different canonical forms [12]. A term of the form $\texttt{let}\ x := t_1\ \texttt{in}\ t_2$ eagerly evaluates t_1 before evaluating t_2.

The Booleans are: $\texttt{tt} = \texttt{inl}(\texttt{Ax})$ and $\texttt{ff} = \texttt{inr}(\texttt{Ax})$. The following operation lifts Booleans to propositions: $\Uparrow(a) = \texttt{tt} \preceq a$, which implies that a is computationally equivalent to $\texttt{tt}$. The following operator asserts that its parameter computes to a value: $\texttt{halts}(t) = \texttt{Ax} \preceq (\texttt{let}\ x := t\ \texttt{in}\ \texttt{Ax})$. We define the following uniform implication: $A \Rrightarrow B = \cap x{:}A.B$, where x does not occur free in B; uniform and: $A \sqcap B = \cap x{:}\texttt{Base}. \cap y{:}\texttt{halts}(x).\texttt{isaxiom}(x, A, B)$; uniform iff: $A \Lleftarrow\!\!\Rrightarrow B = (A \Rrightarrow B \sqcap B \Rrightarrow A)$; computational equivalence: $t_1 \sim t_2 = t_1 \preceq t_2 \sqcap t_2 \preceq t_1$.

Meaning of $\texttt{per}$ types. A term of the form $\texttt{per}(R)$ is a type if for all closed terms t_1 and t_2, $R\ t_1\ t_2$ is a type, and R is a PER on closed terms. Two $\texttt{per}$ types $\texttt{per}(R_1)$ and $\texttt{per}(R_2)$ are equal if for all closed terms t_1 and t_2, $R_1\ t_1\ t_2$ is inhabited iff $R\ t_1\ t_2$ is inhabited. Two terms t_1 and t_2 are equal in $\texttt{per}(R)$ if $R\ t_1\ t_2$ is inhabited. We have formally proved in our Coq metatheory that the derivation rules that implement these conditions are valid [3, Sec. 5.2.4].

Type definitions. We now show how one defines Nuprl's partial and function types using the core type system described above. We first start with the simple $\texttt{Void}$, $\texttt{Unit}$ and $\mathbb{Z}$ types.

$$\texttt{Void} = \texttt{per}(\lambda a.\lambda b.\texttt{tt} \preceq \texttt{ff}) \qquad \texttt{Unit} = \texttt{per}(\lambda a.\lambda b.\texttt{tt} \preceq \texttt{tt})$$

$$\mathbb{Z} = \texttt{per}(\lambda a.\lambda b.a \sim b \sqcap \Uparrow(\texttt{isint}(a, \texttt{tt}, \texttt{ff})))$$

$$a{:}A \to B[a] = \texttt{per}(\lambda f.\lambda g. \cap a, b{:}\texttt{Base}.a = b \in A \Rrightarrow f\ a = g\ b \in B[a])$$

$$\overline{A} = \texttt{per}(\lambda x, y.(\texttt{halts}(x) \Lleftarrow\!\!\Rrightarrow \texttt{halts}(y)) \sqcap (\texttt{halts}(x) \Rrightarrow x = y \in A) \sqcap \cap a{:}\texttt{Base}.a \in A \Rrightarrow \texttt{halts}(a))$$

Using these definitions, several of our inference rules can be proved as lemmas.

Algebraic datatypes. Let $\mathbb{N} = \texttt{per}(\lambda a.\lambda b.a = b \in \mathbb{Z} \sqcap \Uparrow(\texttt{if}\ {-1}{<}a\ \texttt{then}\ \texttt{tt}\ \texttt{else}\ \texttt{ff}))$. We assume the existence of an induction principle on $\mathbb{N}$. Using induction on $\mathbb{N}$ and intersection types, we build coinductive types: $\texttt{corec}(G) = \cap n{:}\mathbb{N}.\texttt{fix}(\lambda P.\lambda n.\texttt{if}\ n{=}0\ \texttt{then}\ \texttt{Top}\ \texttt{else}\ G\ (P\ (n{-}1)))\ n$; and using partial functions and coinductive types we build algebraic datatypes. (In order to build inductive types we can add W types to our core system. However, in a companion paper we discuss how to build inductive types using Bar Induction instead.) Our method consists in selecting the largest collection of terms on which the subterm relation is well-founded. We then derive induction principles using this selection procedure. Given a coalgebraic datatype T, we define a size function s on T. Using fixpoint induction [8] we can prove that for all $t \in T$, $s(t) \in \overline{\mathbb{Z}}$. We can then prove that $(\exists n : \mathbb{N}.\ s(t) = n \in \overline{\mathbb{Z}}) \in \mathbb{P}$. We define our algebraic datatype as $\{t : T \mid \exists n : \mathbb{N}.\ s(t) = n \in \overline{\mathbb{Z}}\}$. To prove inductive properties of algebraic datatypes, we can then go by induction on n.

References

[1] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[2] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006. http://www.nuprl.org/.

[3] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. Technical report, Cornell University, 2014. http://www.nuprl.org/html/verification/.

[4] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. http://coq.inria.fr/.

[5] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd Int'l Conf.*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009. http://wiki.portal.chalmers.se/agda/pmwiki.php.

[6] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[7] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *LICS*, pages 183–193. IEEE Computer Society, 1987.

[8] Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.

[9] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of Fourth IEEE Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, 1989.

[10] Alexei Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects*. PhD thesis, Cornell University, Ithaca, NY, 2004.

[11] Aleksey Nogin and Alexei Kopylov. Formalizing type operations using the "image" type constructor. *Electr. Notes Theor. Comput. Sci.*, 165:121–132, 2006.

[12] Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in Nuprl using computational equivalence and partial types. In *ITP 2013*, volume 7998 of *LNCS*, pages 261–278. Springer, 2013.

Coq à la Tarski: a predicative calculus of constructions with explicit subtyping

Ali Assaf[1][2]

[1] INRIA Paris-Rocquencourt, Paris, France
[2] École Polytechnique, Paris, France

The predicative Calculus of Inductive Constructions (pCIC), the theory behind the Coq proof system, contains an infinite hierarchy of predicative universes

$$Type_0 \in Type_1 \in Type_2 \in \ldots$$

and an impredicative universe *Prop* for propositions, together with an implicit cumulativity relation

$$Prop \subseteq Type_0 \subseteq Type_1 \subseteq Type_2 \subseteq \ldots.$$

This gives rise to a subtyping relation $\leq$ which is used in the subsumption rule

$$\frac{\Gamma \vdash M : A \qquad A \leq B}{\Gamma \vdash M : B}.$$

Subtyping in Coq is implicit, and is handled by the kernel. An attempt to simplify the theory would be to make subtyping explicit, by inserting explicit coercions such as

$$c_{0,1} : Type_0 \to Type_1$$

and rely on a kernel that only uses the classic conversion rule

$$\frac{\Gamma \vdash M : A \qquad A \equiv B}{\Gamma \vdash M : B}.$$

However, because of dependent types, coercions change the shape of the types and therefore interfere with type checking.

We present a formulation of the predicative calculus of constructions using Tarski-style universes [4] where subtyping is explicit. Other such systems have been proposed in the past [5, 2, 3]. However, they do not preserve equality: a term in the original Coq system can have many non-equivalent representations in the new system, which breaks typing. As a result, these systems lose some of the expressivity of Russell-style universes with implicit subtyping, and are therefore incomplete.

Our system fully preserves equality. By adding aditional equations between terms, we ensure that every well-typed term in the original system has a unique canonical representation in our system. To our knowledge, this is the first time such work has been done for the full predicative calculus of constructions. We will also show how to orient the equations into reduction rules. This work can be used as a basis for embedding Coq in a logical framework like the $\lambda\Pi$-calculus modulo [1], implemented in Dedukti [6].

References

[1] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007.

[2] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, Inc., New York, NY, USA, 1994.

[3] Zhaohui 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.

[4] Per Martin-Lof and Giovanni Sambin. *Intuitionistic type theory.* Bibliopolis Naples, 1984.

[5] Erik Palmgren. On universes in type theory. In *Twenty Five Years of Constructive Type Theory.* Oxford University Press, 1998.

[6] Ronan Saillard. Dedukti: a universal proof checker. In *Foundation of Mathematics for Computer-Aided Formalization Workshop*, 2013.

Inductive Construction in Nuprl Type Theory Using Bar Induction

Mark S Bickford & Robert Constable

Cornell University

Contructive type theories such as Coq, Agda, and Nuprl all have some powerful primitive form of inductive construction. The soundness of the rules for these inductive constructions can be difficult to prove. In this note we show that one powerful form of inductive construction, *parameterized families of W-types*, can be internally constructed in type theory using a general form of Brouwer's bar induction rule and induction on a primitive type of natural numbers, from types that need not be defined inductively. We first construct the corecursive family of non-wellfounded types and then construct their wellfounded parts in such a way that the desired induction principle follows from bar induction. All the results have been formally proved in Nuprl, and details can be found here: `http://www.nuprl.org/LibrarySnapshots/Published/Version1/Standard/co-recursion/sbi-param-W-induction.html`.

$S \sqsubseteq T$ means that type S is a subtype of type T. A type function F is monotone if $S \sqsubseteq T \Rightarrow F(S) \sqsubseteq F(T)$, and preserves ω-limits if $\bigcap_{n\in\mathbb{N}} F(X_n) \sqsubseteq F(\bigcap_{n\in\mathbb{N}} X_n)$. A type T is a fixed point of F if $T \sqsubseteq F(T)$ and $F(T) \sqsubseteq T$. For any type T, $T \sqsubseteq Top$, where $Top = \bigcap_{x\in Void} Void$. For any monotone, ω-limit preserving function F, the type $\mathrm{corec}(F) = \bigcap_{n\in\mathbb{N}} F^n(\mathrm{Top})$, where the iteration F^n is defined by primitive recursion, is the greatest fixed point of F. We often write A_p rather than $A(p)$ and $B_{p,a}$ rather than $B(p,a)$.

Parameterized families of co-W and W-types. For parameter type P and functions $A \in P \to \mathrm{Type}$, $B \in p\colon P \to A_p \to \mathrm{Type}$, and $C \in p\colon P \to a\colon A \to B_{p,a} \to P$, the family $W_{A,B,C}(p)$ is the least fixed point of the functional $F_{A,B,C}$ on type families $G \in P \to \mathrm{Type}$ defined by

$$F_{A,B,C}(G) = \lambda p.\ a\colon A \times (b\colon B_{p,a} \to G(C_{p,a,b}))$$

Since $F_{A,B,C}$ is monotone and preserves ω-limits (on type families), we can easily construct the greatest fixed point family, $coW_{A,B,C}$, as follows:

$$coW_{A,B,C} = \lambda p.\ \bigcap_{n\in\mathbb{N}} F^n_{A,B,C}(\lambda q.\ \mathrm{Top})(p)$$

Then, for $S_{A,B,C} = p\colon P \times w\colon coW_{A,B,C} \times (B_{p,\pi_1(w)} + \mathrm{Unit})$, a path has type $\mathrm{Path}^{A,B,C} = \{s\colon \mathbb{N} \to S_{A,B,C} \mid \forall n\colon \mathbb{N}.\ \mathrm{con}(s(n), s(n+1))\}$ where

$$\mathrm{con}(\langle p, \langle a, f\rangle, d_1\rangle, \langle q, w_2, d_2\rangle) \Leftrightarrow (d_1 = \mathrm{inl}(b) \Rightarrow (q = C_{p,a,b} \ \wedge\ w_2 = f(b)))$$

A path s halts, $\mathrm{halts}(s)$, if $\downarrow\exists n\colon \mathbb{N}.\ \exists p.\exists w.\ s(n) = \langle p, w, \mathrm{inr}()\rangle$, where the ***squash*** of a type T is the type $\downarrow T$ that is empty if T is empty and is Unit if T is non-empty. Paths that start at p, w have type

$$\mathrm{Path}^{A,B,C}_{p,w} = \Big\{s\colon \mathrm{Path}^{A,B,C} \mid \exists d.\ s(0) = \langle p, w, d\rangle\Big\}$$

and we define the type $W_{A,B,C}(p)$ by

$$W_{A,B,C}(p) = \Big\{w\colon coW_{A,B,C} \mid \forall s : \mathrm{Path}^{A,B,C}_{p,w}.\ \mathrm{halts}(s)\Big\}$$

It is relatively straightforward to prove that $W_{A,B,C}$ is a fixed point of the functional $F_{A,B,C}$.

To show that it is the least fixed point we use bar induction to prove that its induction principle is witnessed by:

$$\begin{array}{l} \lambda C.\lambda ind.\lambda par.\lambda w.\ \texttt{letrec}\ F(p,w)\ =\ \texttt{let}\ a,f=w\ \texttt{in}\ ind(p,a,f,\lambda b.F(C[p;a;b],f(b)) \\ \qquad\qquad\qquad\qquad\ \texttt{in}\ F(par;w) \end{array}$$

Bar Induction. A finite sequence s of length k has type $V_k(T) = \mathbb{N}_k \to T$, and we append t to s using $s \oplus_k t = \lambda i.$ if $i < k$ then $s(i)$ else t. Our bar induction rule is restricted to conclusions of the form $a(k,s) \in X(k,s)$, which, in Nuprl, have trivial constructive content. Let $ind(R,T,a,X,k,s,t)$ be the formula

$$\forall t\colon \{t\colon T \mid R(k,s,t)\}\,.\,a(k+1,s\oplus_k t) \in X(k+1,s\oplus_k t)$$

The (restricted) bar induction rule is:

$$\frac{\begin{array}{c} H \vdash T \in \text{Type} \qquad H,\ k\colon \mathbb{N},\ s\colon V_k(T),\ t\colon T \vdash R(k,s,t) \in \text{Type} \\ H,\ k\colon \mathbb{N}\,,s\colon V_k(T),\ con(R,k,s) \vdash B(k,s)\ \vee\ \neg B(k,s) \\ H,\ f\colon \mathbb{N} \to T,\ \forall i\colon \mathbb{N}.\ R(i,f,f(i)) \vdash\downarrow \exists n\colon \mathbb{N}.\ B(n,f) \\ H,\ k\colon \mathbb{N}\,,s\colon V_k(T),\ con(R,k,s),\ B(k,s) \vdash a(k,s) \in X(k,s) \\ H,\ k\colon \mathbb{N}\,,s\colon V_k(T),\ con(R,k,s),\ ind(R,T,a,X,k,s,t) \vdash a(k,s) \in X(k,s) \end{array}}{H \vdash a(0,z) \in X(0,z)}$$

The first two premises give the type of the spread law R. The next two premises state that B is a decidable bar on the spread defined by R. The fifth and sixth premises are the base and induction steps of the proof by bar induction for the term $a(0,z) \in X(0,z)$ in the conclusion of the rule. This is a strong form of bar induction because the spread law R can be any relation not necessarily decidable.

Since Nuprl allows general recursive definitions we can define bar recursion as

$$\text{br}(d,b,i,n,s) = \text{if } d{=}\text{inl}(x) \text{ then } b(n,s,x) \text{ else } i(n,s,\lambda t.\ \text{br}(d,b,i,n+1,s\oplus_n t))$$

and, using the restricted bar induction rule, we prove that bar recursion is the realizer for the general, unrestricted form of bar induction.

Remarks.

1. As described in a companion paper, all the non-inductive types can be built using only three type constructors, intersection, equality, and PER, which forms a type from a partial equivalence relation on closed terms.
2. Anand and Rahli have implemented Nuprl in Coq by defining its computation system, type system, sequents and rules. The type system they define has W types as primitives and does not include Mendler's recursive types. They have both an impredicative model of all the universes and a predicative model of finitely many.
3. Nuprl currently uses Mendler's recursive types, but every use of a recursive type in our library could be replaced with a W-type.
4. The results in this paper reduces the soundness of inductive constructions to the soundness of the bar induction rule given above. Because bar induction is true in classical logic, we should be able to prove it in the impredicative Coq model of Nuprl using the excluded middle axiom. This is work in progress.
5. We believe that analogues of Coq's inductive types can be defined using parameterized W-types because Nuprl's type theory satisfies function extensionality.
6. We do not know whether Agda's inductive-recursive constructions can be defined using the method of this paper (corecursion and bar induction).

Eliminating Higher Truncations via Constancy

Paolo Capriotti and Nicolai Kraus

University of Nottingham

Abstract

We show how to construct functions $\|A\|_n \to B$ if B is not an n-type.

In Homotopy Type Theory (HoTT), *truncations* constitute an important class of *higher inductive types*: for any type A and any integer $n \geq -1$, the n-truncation $\|A\|_n$ can be understood as a version of the type A where all homotopical structure above level n is collapsed. In general, a type without any nontrivial structure above level n is called an n-type.

When $n = -1$, truncations correspond to the *squashing* or *bracketing* [1] operator, of which they can be thought of as higher-dimensional generalisations. Truncations are usually presented as reflectors of the corresponding sub-$(\infty, 1)$-categories of n-types, resulting in an elimination principle which only allows n-types as targets. Given a function $f : A \to B$, we can construct a function $\|A\|_n \to B$ as long as B is an n-type.

If B happens to be an m-type for some $m > n$, then the eliminator cannot be applied directly. Therefore, in order to factor a function $f : A \to B$ through the truncation $\|A\|_n$, the usual approach is to construct an ad-hoc n-type P, and show, with the help of the eliminator, that f factors through P. However, it is not always clear how to construct such a type P.

We address this problem in vast generality by reducing the problem of factoring a function $f : A \to B$ through $\|A\|_n$ to that of proving a number of coherence conditions on f.

The simplest nontrivial special case of our construction is, for a given 0-truncated type B (a *set*), the equivalence

$$\left(\|A\|_{-1} \to B\right) \;\simeq\; \left(\Sigma_{f:A\to B}\, \forall a_1\, a_2.\, f(a_1) = f(a_2)\right). \tag{1}$$

This equivalence tells us that, in order to define $\|A\|_{-1} \to B$, we need to find $f : A \to B$ and a proof that f takes equal values for any pair of points in its domain. The latter can be understood as a weak form of constancy. Let us write C_1 for this condition:

$$C_1^{A;f} \;:\equiv\; \Pi_{a_1,a_2:A}\, f(a_1) = f(a_2).$$

Unfortunately, the equivalence (1) breaks down when B is anything other than a 0-type (a related explanation can be found in [3]). For example, if B is a 1-type, then, given a function $f : A \to B$, a term $c_1 : C_1^{A;f}$ is not sufficient to guarantee that f factors through $\|A\|_{-1}$. We need to impose an additional condition: c_1 should provide "coherent" equality proofs in B. More precisely, we require an inhabitant of the type

$$C_2^{A;f,c_1} \;:\equiv\; \Pi_{a_1,a_2,a_3:A}\, c_1(a_1,a_2) \cdot c_1(a_2,a_3) = c_1(a_1,a_3).$$

Indeed, we can then prove the following equivalence: for any type A and any 1-type B,

$$\left(\|A\|_{-1} \to B\right) \;\simeq\; \left(\Sigma_{f:A\to B}\, \Sigma_{c_1:C_1^{A;f}}\, C_2^{A;f,c_1}\right). \tag{2}$$

We can deal with higher truncations similarly. For example, factoring through the 0-truncations requires the same conditions, but this time they are imposed on ap_f rather than f directly. When B is a 1-type, we then obtain the equivalence:

$$\left(\|A\|_0 \to B\right) \;\simeq\; \left(\Sigma_{f:A\to B}\, \Pi_{a:A} C_1^{\Omega(A,a);\mathsf{ap}_f}\right). \tag{3}$$

It is not hard to imagine that the equivalences (1), (2) and (3) can be generalised both to any truncation level of B and to any truncation operator, by formulating appropriate coherence conditions C_n for all n, of which C_1 and C_2 are the first two examples.

For any fixed numbers $k, n \geq -1$ (technically, -2) and n-type B, we show how to construct a type in basic HoTT (using Σ, Π, Id only) which is equivalent to $\|A\|_k \to B$. Our proof of the equivalence, however, requires higher inductive types for $k \geq 0$.

We view these equivalences as *generalised universal properties* of the truncations. In the case $n = k$, they degenerate to the ordinary universal property of the truncation, as defined in [4, Lemma 7.3.3], $(\|A\|_n \to B) \;\simeq\; (A \to B)$.

Note that it is not immediately obvious how to even express the general result. It is not difficult to guess the conditions C_n for the first few values of n, but, although the pattern is intuitively clear, it is quite hard to capture it precisely.

Fortunately, Shulman's work on inverse diagrams [5] provides a powerful framework, which helps formulate and reason about towers of coherence conditions.

The idea to generalise the equivalences (1) and (2) to higher truncation levels of B (but still assuming $k = -1$ for now) is that the required coherence conditions may be regarded as a morphism between two *semi-simplicial types* [2]: the 0-coskeletal semi-simplicial type "generated" by A, and the *equality semi-simplicial type on B* (an explicit Reedy-fibrant resolution of B regarded as a constant presheaf). The statement of the general case ($k \geq -1$) can then be obtained by applying the above construction to ap_f^{k+1}.

We state and prove our result in any model of HoTT with ω^{op}-Reedy limits, without putting any restriction on B. However, if B is an n-type for some n, as is often the case for the target of an eliminator, the condition of existence of Reedy limits in the model can be dropped (intuitively, the infinite tower of conditions becomes finite).

In that case, we obtain a simplified formulation, which, for any fixed pair of natural numbers k and n, holds in *any* model of HoTT. In particular, this is true for the initial (syntactical) model, and the equivalences can be used when formalising mathematics in a proof assistant.

Note that our result can simply be regarded as a family of types and equivalences depending on the two indices k and n. The construction is uniform enough to be done mechanically, that is, one could write a program which takes k, n as inputs and generates the required types and proof terms. However, we believe that it is impossible to perform this construction internally for *variables* k, n. This is closely related to the difficulties encountered by several people when trying, for example, to formalise HoTT in itself; Shulman has recently given a thorough analysis of this phenomenon [6].

References

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

[2] Hugo Herbelin. A dependently-typed construction of semi-simplicial types. 2014.

[3] Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of anonymous existence in Martin-Löf type theory. 2014. In preparation.

[4] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. first edition, 2013. Available online at homotopytypetheory.org/book.

[5] Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *ArXiv e-prints*, March 2012.

[6] Michael Shulman. Homotopy type theory should eat itself (but so far, its too big to swallow). Blog post at homotopytypetheory.org, March 2014.

Objects and subtyping in the $\lambda\Pi$-calculus modulo

Ali Assaf[1][2], Raphaël Cauderlier[1][3], and Catherine Dubois[3][4]

[1] INRIA Paris-Rocquencourt, Paris, France
[2] École Polytechnique, Paris, France
[3] CNAM, Paris, France
[4] ENSIIE, Évry, France

In this talk, we present a shallow embedding of an object calculus, the ς-calculus, in the $\lambda\Pi$-calculus modulo. The main difficulty is the encoding of subtyping. We propose a solution that makes use of rewriting in order to ease the handling of subtyping proofs.

Motivations The $\lambda\Pi$-calculus modulo is an extension of the $\lambda\Pi$-calculus [7] with rewrite rules. Implemented in the Dedukti type-checker [9], it can be used as a logical framework for the implementation of formal systems [5]. In this framework, rewrite rules can be introduced in addition to β-reduction to extend the conversion relation between terms.

Cousineau and Dowek [4] showed that any functional pure type system can be encoded in the $\lambda\Pi$-calculus modulo using appropriate rewrite rules. The main emphasis of this embedding is that it is *shallow*, as opposed to *deep* embeddings. As much as possible, the features of the object language are implemented by the corresponding features in the meta-language: bindings are represented using binders, typing using typing, reduction using reduction, etc. Besides avoiding the reimplementation of these features, shallow embeddings have the advantage of being more compact, more readable, and more efficient than deep embeddings.

While encoding languages with functional features in the $\lambda\Pi$-calculus modulo seems natural, encoding object-oriented languages, that share no feature with the $\lambda\Pi$-calculus modulo, is less obvious. In particular, encoding subtyping is a challenging problem, because it is absent from the target language. In the $\lambda\Pi$-calculus modulo, each term has a unique type. If M has type A and A is not convertible to B then M does not have type B. Moreover, it is not possible to rewrite A to B, as then any term of type B would also have type A, which would be unsound.

Related work A lot of work has been done in the field of encoding of objects. Several such encodings in System $F^{\omega}_{<:}$ have been proposed and compared [8, 2]. They often rely on existential types and some form of recursion.

In 1996, Abadi and Cardelli [1] defined several object calculi which consider objects as a primitive notion instead of encoding them in a λ-calculus. These calculi are very primitive in the sense that they can be used to represent both object-based and class-based languages and they do not distinguish methods from fields. These calculi have been used as examples for testing the effectivity of deep embeddings in the Coq proof assistant [6, 3].

One of these calculi is the simply-typed ς-calculus. It represents objects as records of the form $[l_i = \varsigma(x : A)t_i]_{i=1\dots n}$, and each method has only one parameter, introduced by the ς binder, which represents *self*. This calculus has simple typing rules and operational semantics.

$$\frac{\Gamma, x : A \vdash t_i : A_i \quad \forall i = 1 \dots n}{\Gamma \vdash t : A} \quad \text{where } A = [l_i : A_i] \text{ and } t = [l_i = \varsigma(x : A)t_i]_{i=1\dots n}$$

$$\begin{array}{lcll} t.l_j & \longrightarrow & t_j\{x := t\} & \text{(method selection)} \\ t.l_j \Leftarrow \varsigma(x : A)u & \longrightarrow & t\ \{l_j := \varsigma(x : A)u\} & \text{(method update)} \end{array}$$

Subtyping is defined by $[l_i : A_i]_{i=1\ldots n+m} <: [l_i : A_i]_{i=1\ldots n}$, so A is a subtype of B if and only if A and B coincide on the labels of B. With its minimalist definition, the simply-typed ς-calculus is an ideal candidate for the study of encodings of object-oriented mechanisms.

Contributions We give an encoding of the simply-typed ς-calculus in the λΠ calculus modulo. We encode types and objects using association lists. Since sub-lists of well-typed objects need not be well-typed, we have to introduce partially constructed (ill-typed) objects. Selection and update of methods are performed using

$$\begin{array}{lll}\mathsf{select} & : & \Pi A : \mathsf{type}, \mathsf{Object}\ A \to \Pi l : \mathsf{Label}, \mathsf{Object}\ (\mathsf{assoc}\ A\ l) \\ \mathsf{update} & : & \Pi A : \mathsf{type}, \mathsf{Object}\ A \to \Pi l : \mathsf{Label}, (\mathsf{Object}\ A \to \mathsf{Object}\ (\mathsf{assoc}\ A\ l)) \to \mathsf{Object}\ A\end{array}$$

and the operational semantics is translated to the following rewrite rules:

$$\begin{array}{llll}\mathsf{select} & A\ [l = m, \ldots]\ l & \hookrightarrow & m\ [l = m, \ldots] \\ \mathsf{update} & A\ [l = m, \ldots]\ l\ m' & \hookrightarrow & [l = m', \ldots]\end{array}$$

We use explicit coercions to handle subtyping. The coercion function

$$\mathsf{coerce} : \Pi A, B : \mathsf{type}, \mathsf{proof}(A <: B) \to \mathsf{Object}\ A \to \mathsf{Object}\ B$$

takes an extra logical argument of type $\mathsf{proof}(A <: B)$. We show how to make special use of rewrite rules and reflection to reduce $\mathsf{proof}(A <: B)$ to $\mathsf{proof}\ \top$ and thus avoid carrying big subtyping proofs.

This encoding has been implemented in Dedukti and tested on the examples from Abadi and Cardelli [1] which illustrate all the features of the simply-typed ς-calculus. Our implementation can be found online at: `https://www.rocq.inria.fr/deducteam/Sigmaid`.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer New York, 1996.

[2] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

[3] A. Ciaffaglione, L. Liquori, and M. Miculan. Reasoning about object-based calculi in (co)inductive type theory and the theory of contexts. *J. Autom. Reasoning*, 39(1):1–47, 2007.

[4] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[5] G. Dowek. A theory independent curry-de bruijn-howard correspondence. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II*, ICALP'12, pages 13–15, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] G. Gillard. A formalization of a concurrent object calculus up to alpha-conversion. In D. A. McAllester, editor, *CADE*, volume 1831 of *LNCS*, pages 417–432. Springer, 2000.

[7] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[8] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[9] R. Saillard. Dedukti: a universal proof checker. In *Foundation of Mathematics for Computer-Aided Formalization Workshop*, Padova, 2013.

Pattern matching without K

Jesper Cockx, Dominique Devriese, and Frank Piessens

DistriNet – KU Leuven

Dependent pattern matching [Coquand, 1992] is a technique for writing functions in languages based on dependent type theory, such as Agda [Norell, 2007], Coq [Sozeau, 2010], and Idris [Brady, 2013]. It allows us to define functions in a style similar to functional programming languages such as Haskell. Additionally, dependent pattern matching can be used to write *proofs* in the form of dependently-typed functions. For example, we can prove the transitivity of the propositional equality $x \equiv y$ by pattern matching on its only constructor $\texttt{refl} : x \equiv x$:

$$\begin{aligned} &\texttt{trans} : (x\ y\ z : A) \to x \equiv y \to y \equiv z \to x \equiv z \\ &\texttt{trans}\ x\ \lfloor x \rfloor\ \lfloor x \rfloor\ \texttt{refl}\ \texttt{refl} = \texttt{refl} \end{aligned} \tag{1}$$

Inaccessible patterns, like $\lfloor x \rfloor$ in this example, witness the fact that only one type-correct argument can be in that position. Indeed, matching on a proof of $x \equiv y$ with $\texttt{refl} : x \equiv x$ forces x and y to be the same.

Proofs by dependent pattern matching are typically much shorter and more readable than ones that use the classical *datatype eliminators* associated to each inductive family. On the other hand, Goguen et al. [2006] showed that all definitions by dependent pattern matching can be translated to ones that only use eliminators. For this translation they depend on the so-called K axiom. Coquand [1992] already observed that pattern matching allows proving this K axiom:

$$\begin{aligned} \mathrm{K} : {}& (P : a \equiv a \to \mathit{Set}) \to \\ & (p : P\ \texttt{refl})(e : a \equiv a) \to P\ e \\ \mathrm{K}\ \ & P\ p\ \texttt{refl} = p \end{aligned} \tag{2}$$

An emerging field within dependent type theory is *homotopy type theory* (HoTT) [The Univalent Foundations Program, 2013]. It gives a new interpretation of terms of type $x \equiv y$ as *paths* from x to y. Many basic constructions in HoTT can be written very elegantly using pattern matching, for example $\texttt{trans}$ (1) corresponds to the composition of two paths.

One of the core elements of HoTT is the *univalence axiom*. Univalence captures the common mathematical practice of informal reasoning "up to isomorphism" in a nice and formalized way. It also has a number of useful consequences, such as *functional extensionality*. However, the univalence axiom is *incompatible* with dependent pattern matching. This has forced people working on HoTT to avoid using pattern matching or risk unsoundness.

The source of the incompatibility between univalence and dependent pattern matching is that pattern matching relies on the K axiom. In an attempt to fix this, an option called –without-K was added to Agda. In theory, this option should allow people to use pattern matching in a safe way when it is undesirable to assume K. However, the option has been criticized many times for being too restrictive, for having unclear semantics, and for containing errors. These errors allowed one to prove (weaker versions of) the K axiom. While they are typically fixed quickly, this really calls for a more in-depth investigation of dependent pattern matching without K.

We present a new criterion that describes what kind of definitions by pattern matching are still allowed if we do not assume K, which is strictly more general than previous attempts. Our criterion works by limiting the unification algorithm used for case splitting in two ways:

- It is not allowed to delete equations of the form $x = x$.

- When applying injectivity on the equation $c\,\bar{s} = c\,\bar{t}$ where $c\,\bar{s}, c\,\bar{t} : D\,\bar{u}$, the indices $\bar{u}$ (but not the parameters) should be *self-unifiable*, i.e. unification of $\bar{u}$ with itself should succeed positively (while still adhering to these two restrictions).

This criterion has been implemented as a patch to Agda. It allows the definition of `trans` (1), but it prohibits the definition of K (2) because case splitting on the argument of type $a \equiv a$ fails.

We give a formal proof that definitions by pattern matching satisfying this criterion are conservative over standard type theory by translating them to eliminators in the style of Goguen et al. [2006], *without* relying on the K axiom. Our proof follows the same general outline, but there are two important differences:

- We work with the homogeneous equality instead of the heterogeneous version, because the elimination rule for the heterogeneous equality is equivalent with K [McBride, 2000].
- Working with the homogeneous equality leads us naturally to upgraded versions of the unification transitions given by Goguen et al. [2006], where the return type is dependent on the equality proof.

We hope that this is enough to convince the HoTT community that pattern matching *can* be used safely without assuming K, and maybe even helps in the creation of a language based on HoTT. Our criterion makes it possible to do pattern matching on *regular* inductive families without assuming K. But HoTT also introduces the concept of *higher inductive types*, which can have nontrivial identity proofs between their constructors. This implies that in general they don't satisfy the injectivity, disjointness, or acyclicity properties. Luckily, our proof is entirely *parametric* in the actual unification transitions that are used. So in order to allow pattern matching in a context with higher inductive types, we should just limit the unification algorithm further.

References

Edwin Brady. Idris, a general purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5), 2013.

Thierry Coquand. Pattern matching with dependent types. In *Types for proofs and programs*, 1992.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*. 2006.

Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 2000.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In *Interactive theorem proving*, 2010.

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

Type-Checking Linear Dependent Types

Arthur Azevedo de Amorim[1], Emilio Jesús Gallego Arias[1], Marco Gaboardi[2], and Justin Hsu[1]

[1] University of Pennsylvania
[2] University of Dundee

Linear indexed type systems have been used to ensure safety properties of programs with respect to different kinds of resources; examples include usage analysis [10], implicit complexity [3], and more. Linear indexed types use a type-level *index language* to describe resources and *linear types* to reason about the program's resource usage in a compositional way.

A limitation of current analysis techniques for such systems is that resource usage is inferred independently of the control flow of a program—e.g., the typing rule for branching usually approximates resources by taking the maximal usage of one of the branches. To make this analysis more precise, some authors have proposed extending adding dependent types, considering both resource usage and the *size information* of a program's input. This significantly enriches the resulting analysis by allowing resource usage to depend on runtime information. Linear dependent type systems have been used in several domains, such as implicit complexity [1] and others.

Of course, there is a price to be paid for the increase in expressiveness: type checking and type inference inevitably become more complex. In linear indexed type systems, these tasks are often done in two stages: a standard Hindley-Milner-like pass, followed by a constraint-solving procedure. In some cases, the generated constraints can be solved automatically with custom algorithms [6] or off-the-shelf SMT solvers [4]. However, the constraints are specific to the index language, and richer index languages often lead to more complex constraints.

In this work we consider the type-checking problem for a particular system with linear dependent types, *DFuzz*. *DFuzz* was born out of *Fuzz* [9], a language where types are used to reason about sensitivity of programs, which measures the distance between outputs on nearby inputs. *Fuzz* uses real numbers as indices for the linear types, which provide an upper bound on the sensitivity of the program. As shown by [4], type-checking *Fuzz* programs can be done efficiently by using an SMT solver to discharge the numeric proof obligations arising from the type system. The same approach works for type inference, which infers the minimal sensitivity of a function.

DFuzz [5] was introduced to overcome a fundamental limitation of *Fuzz*: sensitivity information cannot depend on runtime information, such as the size of a data structure. This is done by enriching *Fuzz* with a limited form of dependent types, whose index language combines information about the *size* of data structures and the *sensitivity* of functions. These changes have a significant impact on the difficulty of type checking, since type checking constraints in *DFuzz* may involve general polynomials rather than just constants.

One solution could be to extend the algorithm proposed in [4] to work with the new index language by generating additional constraints when dealing with the new constructs. This would be similar in spirit to the work of [2] for type inference for $\mathsf{d}\ell\mathsf{PCF}$, a linear dependent type system for complexity analysis. Unfortunately, such an approach does not work as well for *DFuzz*, since it relies on the presence of arbitrary computable functions in the index language, whereas *DFuzz*'s index language is far simpler. Instead, since the type system of *DFuzz* also supports subtyping, we consider a different approach inspired by techniques from the literature on subtyping (e.g. [7]) and on constraint-based type-inference approaches (e.g. [8]).

The main idea is to type-check a program by inferring some set of sensitivities for it, and then testing whether the resulting type is a subtype of the desired type. To obtain completeness, one must ensure that the inferred sensitivities are the “best” possible. Unfortunately, the *DFuzz* index language is not rich enough for expressing such sensitivities. For instance, some cases require taking the maximum of two sensitivity expressions, which may not lie inside the basic sensitivity language. We solve this problem by extending the index language with a handful of index constructs to ease sensitivity-inference; we call this new system *EDFuzz*. We present a sensitivity-inference algorithm for *EDFuzz*, which we show sound and complete. Furthermore, *EDFuzz* has similar meta-theoretic properties as *DFuzz*.

We are left with the problem of solving the constraints generated by our algorithm. First, we show how to compile the constraints generated by the algorithmic systems to first-order constraints, allowing us to use standard solvers. Unfortunately, the resulting set of constraints is too powerful, and we also show that type checking for *DFuzz* is undecidable. We discuss how to approximate complete type-checking with a constraint relaxation procedure that is enough to handle the examples proposed in [5].

References

[1] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *IEEE Symposium on Logic in Computer Science (LICS), Toronto, Ontario*, pages 133–142. IEEE, 2011.

[2] Ugo Dal Lago, Barbara Petit, et al. The geometry of types. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 167–178, 2013.

[3] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In *ACM Transactions on Programming Languages and Systems*, pages 205–225. Springer, 2010.

[4] Loris D'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin C. Pierce. Sensitivity analysis using type-based constraints. In *Workshop on Functional Programming Concepts in Domain-specific Languages (FPCDSL)*, FPCDSL '13, pages 43–50, New York, NY, USA, 2013. ACM.

[5] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, POPL '13, pages 357–370, New York, NY, USA, 2013. ACM.

[6] Ugo Dal Lago and Ulrich Schöpp. Type inference for sublinear space functional programming. In Kazunori Ueda, editor, *Asian Symposium on Programming Languages and Systems (APLAS), Shanghai, China*, volume 6461 of *Lecture Notes in Computer Science*, pages 376–391. Springer, 2010.

[7] Benjamin C. Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, pages 511–530, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[8] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[9] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, ICFP '10, pages 157–168, New York, NY, USA, 2010.

[10] Philip Wadler. Is there a use for linear logic? In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), New Haven, Connecticut*, volume 26, pages 255–273. ACM, 1991.

Simply Typed Lambda-Calculus Modulo Type Isomorphisms [*]

Alejandro Díaz-Caro[1,2] and Gilles Dowek[2]

[1] Université Paris-Ouest Nanterre La Défense, 200 avenue de la République, 92001 Nanterre, France
[2] INRIA, 23 avenue d'Italie, CS 81321, 75214 Paris Cedex 13, France

In informal mathematics, isomorphic structures are often identified. For instance, the natural numbers and non negative integers are never distinguished, although they formally are different structures.

In typed lambda-calculus, in programming languages, and in proof theory, two types A and B are said to be isomorphic, when there exists two functions ϕ from A to B and ψ from B to A such that $\psi\phi\mathbf{r} = \mathbf{r}$ for all terms $\mathbf{r}$ of type A and $\phi\psi\mathbf{s} = \mathbf{s}$ for all terms $\mathbf{s}$ of type B.

In some cases, isomorphic types are identified. For instance, in Martin-Löf's type theory [16], in the Calculus of Constructions [8], and in Deduction modulo [14,15], definitionally equivalent types are identified. For example, the types $x \subseteq y$, $x \in \mathcal{P}(y)$ and $\forall z\ (z \in x \Rightarrow z \in y)$ may be identified. However, definitional equality does not handle all the isomorphisms and, for example, the isomorphic types $A \wedge B$ and $B \wedge A$ are not usually identified: a term of type $A \wedge B$ does not have type $B \wedge A$.

Not identifying such types has many drawbacks, for instance if a library contains a proof of $B \wedge A$, a request on a proof of $A \wedge B$ fails to find it [18], if $\mathbf{r}$ and $\mathbf{s}$ are proofs of $(A \wedge B) \Rightarrow C$ and $B \wedge A$ respectively, it is not possible to apply $\mathbf{r}$ to $\mathbf{s}$ to get a proof of C, but we need to explicitly apply a function of type $(B \wedge A) \Rightarrow (A \wedge B)$ to $\mathbf{s}$ before we can apply $\mathbf{r}$ to this term. This has lead to several projects aiming at identifying in a way or another isomorphic types in type theory, for instance with the univalence axiom [4]. Identifying types also leads, as we shall see, to interesting calculi.

In [6], Bruce, Di Cosmo and Longo have provided a characterisation of isomorphic types in the simply typed λ-calculus extended with products and a unit type. In this work, we fully defined a simply typed λ-calculus extended with products, where all the isomorphic types are identified, and we provide a proof of strong normalisation for it. All the isomorphisms in such a setting, can be summarised to the following four:

$$A \wedge B \equiv B \wedge A \qquad (1) \qquad\qquad A \Rightarrow (B \wedge C) \equiv (A \Rightarrow B) \wedge (A \Rightarrow C) \qquad (3)$$

$$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C \qquad (2) \qquad\qquad (A \wedge B) \Rightarrow C \equiv A \Rightarrow B \Rightarrow C \qquad (4)$$

Any other isomorphisms can be obtained by a combination of the previous four. For example, $A \Rightarrow B \Rightarrow C \equiv B \Rightarrow A \Rightarrow C$ is a consequence of isomorphisms (4) and (1).

Identifying types requires to also identify terms. For example, if $\langle \mathbf{r}, \mathbf{s} \rangle$ has type $A \wedge B = B \wedge A$, then it is not clear what the first projection would be. A more elaborated example, if $\mathbf{r}$ is a closed term of type A, then $\lambda x^A.x$ is a term of type $A \Rightarrow A$, and $\langle \lambda x^A.x, \lambda x^A.x \rangle$ is a term of type $(A \Rightarrow A) \wedge (A \Rightarrow A)$, hence, by isomorphism (3), a term of type $A \Rightarrow (A \wedge A)$. Thus the term $\langle \lambda x^A.x, \lambda x^A.x \rangle \mathbf{r}$ is a term of type $A \wedge A$. Although this term contains no redex, we do not want to consider it as normal, in particular because it is not an introduction. So we shall distribute the application over the comma, yielding the term $\langle (\lambda x^A.x)\mathbf{r}, (\lambda x^A.x)\mathbf{r} \rangle$ that finally reduces to $\langle \mathbf{r}, \mathbf{r} \rangle$. Similar considerations lead to introduction of several equivalence rules on terms.

[*] Full version of this paper available at http://www.diaz-caro.info/stmti.pdf (submitted)

One of the main difficulties in the design of this calculus is the design of the elimination rule for the conjunction. A rule like "if $\mathbf{r} : A \wedge B$ then $\pi_1(\mathbf{r}) : A$", would not be consistent. Indeed, if A and B are two arbitrary types, $\mathbf{s}$ a term of type A and $\mathbf{t}$ a term of type B, then $\langle \mathbf{s}, \mathbf{t} \rangle$ has both types $A \wedge B$ and $B \wedge A$, thus $\pi_1 \langle \mathbf{s}, \mathbf{t} \rangle$ would have both type A and type B. The approach we have followed is to consider explicitly typed (Church style) terms, and parametrise the projection by the type: if $\mathbf{r} : A \wedge B$ then $\pi_A(\mathbf{r}) : A$ and the reduction rule is then that $\pi_A \langle \mathbf{s}, \mathbf{t} \rangle$ reduces to $\mathbf{s}$ if $\mathbf{s}$ has type A.

But this rule introduces some non-determinism. Indeed, in the particular case where A happens to be equal to B, then both $\mathbf{s}$ and $\mathbf{t}$ have type A and $\pi_A \langle \mathbf{s}, \mathbf{t} \rangle$ reduces both to $\mathbf{s}$ and to $\mathbf{t}$. Notice that although this reduction rule is non-deterministic, it preserves typing.

Thus, our calculus is one of the many non-deterministic calculi in the line of [5,7,9,10,12,17]. Our pair-construction operator is like the parallel composition operator in a non deterministic calculi. In non-deterministic calculi, the parallel composition is such that if $\mathbf{r}$ and $\mathbf{s}$ are two λ-terms, the term $\langle \mathbf{r}, \mathbf{s} \rangle$ represents the computation that runs either $\mathbf{r}$ or $\mathbf{s}$ non-deterministically, that is such that $\langle \mathbf{r}, \mathbf{s} \rangle \mathbf{t}$ reduces either to $\mathbf{rt}$ or $\mathbf{st}$. In our case, $\pi_B(\langle \mathbf{r}, \mathbf{s} \rangle \mathbf{t})$ first reduces to $\pi_{A \Rightarrow B} \langle \mathbf{r}, \mathbf{s} \rangle \mathbf{t}$ and then to $\mathbf{rt}$ or $\mathbf{st}$.

In [11] we showed that this calculus is also related to the algebraic calculi [1–3,13,19], some of which have been designed to express quantum algorithms. In this case, the pair $\langle \mathbf{s}, \mathbf{t} \rangle$ is not interpreted as a non deterministic choice but as a superposition of two processes running $\mathbf{s}$ and $\mathbf{t}$, and the projection π is the related to the projective measurement, that is the only non deterministic operation.

References

[1] P. Arrighi and A. Díaz-Caro. A System F accounting for scalars. *LMCS*, 8(1:11), 2012.

[2] P. Arrighi, A. Díaz-Caro, and B. Valiron. A type system for the vectorial aspects of the linear-algebraic lambda-calculus. *EPTCS*, 88:1–15, 2012.

[3] P. Arrighi and G. Dowek. Linear-algebraic lambda-calculus: higher-order, encodings, and confluence. *LNCS*, 5117:17–31, 2008.

[4] S. Awodey, A. Pelayo, and M. A. Warren. Voevodsky's univalence axiom in homotopy type theory. *Notices of the AMS*, 60(08):1164–1167, 2013.

[5] G. Boudol. Lambda-calculi for (strict) parallel functions. *IC*, 108(1):51–127, 1994.

[6] K. B. Bruce, R. Di Cosmo, and G. Longo. Provable isomorphisms of types. *MSCS*, 2(2):231–247, 1992.

[7] A. Bucciarelli, T. Ehrhard, and G. Manzonetto. A relational semantics for parallelism and non-determinism in a functional setting. *Annals of Pure and Applied Logic*, 163(7):918–934, 2012.

[8] T. Coquand and G. Huet. The calculus of constructions. *IC*, 76(2–3):95–120, 1988.

[9] U. de'Liguoro and A. Piperno. Non deterministic extensions of untyped λ-calculus. *IC*, 122(2):149–177, 1995.

[10] M. Dezani-Ciancaglini, U. de'Liguoro, and A. Piperno. A filter model for concurrent lambda-calculus. *SIAM Journal of Computing*, 27(5):1376–1419, 1998.

[11] A. Díaz-Caro and G. Dowek. The probability of non-confluent systems. *EPTCS*, 143:1–15, 2013.

[12] A. Díaz-Caro, G. Manzonetto, and M. Pagani. Call-by-value non-determinism in a linear logic type discipline. *LNCS*, 7734:164–178, 2013.

[13] A. Díaz-Caro and B. Petit. Linearity in the non-deterministic call-by-value setting. *LNCS*, 7456:216–231, 2012.

[14] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.

[15] G. Dowek and B. Werner. Proof normalization modulo. *The Journal of Symbolic Logic*, 68(4):1289–1316, 2003.

[16] P. Martin-Löf. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, 1984.

[17] M. Pagani and S. Ronchi della Rocca. Linearity, non-determinism and solvability. *Fundamenta Informaticae*, 103(1-4):173–202, 2010.

[18] M. Rittri. Retrieving library identifiers via equational matching of types. *LNCS*, 449:603–617, 1990.

[19] L. Vaux. The algebraic lambda calculus. *MSCS*, 19(5):1029–1059, 2009.

Synthesis of certified programs with effects using monads in Coq

Sara Fabbro and Marino Miculan

Laboratory of Models and Applications of Distributed Systems
Department of Mathematics and Computer Science, University of Udine, Italy
`fabbro.sara.88@gmail.com`, `marino.miculan@uniud.it`

An important feature of type-theory based proof assistants is the possibility of extracting *certified* programs from proofs [4, 2], in virtue of the Curry-Howard "proofs-as-programs", "propositions-as-types" isomorphism. The extracted programs are certified in the sense that they are guaranteed to satisfy their *specification*, i.e. the properties represented by their types in the proof assistant.

One limitation of this approach is that these programs are always purely functional. Non-functional programming languages (e.g. imperative, distributed, concurrent,...) hardly feature a type theory supporting a Curry-Howard isomorphism, and even if such a theory were available, implementing a specific proof-assistant with its own extraction facilities would be a daunting task.

In this talk we present a methodology for circumventing this problem using the extraction mechanisms of existing proof assistants (namely Coq), generalizing previous work [3]. Basically, the idea is to incapsulate the non-functional aspects in a *computational monad*, as done in Haskell, and using the extraction facilities of Coq for directly producing certified Haskell code with monads.

Let us consider an algebraic specification (T, Σ, Γ) of a monad T. This consists of an abstract type constructor T (i.e., for each type A, TA is the type of computations whose values have type A), and a set $\Sigma = \{op_1, \dots, op_n\}$ of (multi-sorted) constructors for the monadic types TA. The behaviour of these constructors is specified by the set $\Gamma = \{s_1 = t_1, \dots, s_m = t_m\}$ of equational laws; terms s_i, t_i in these equations are built using the operators in Σ, plus the basic constructors of any monad $return_A : A \to TA$ and $bind_{A,B} : TA \to (A \to TB) \to TB$ (often written $\gg=$). For instance, the "maybe" monad M is defined by a single (polymorphic) constructor $nothing_A : MA$, and a single equation $bind_{A,B}(nothing_A, f) = nothing_B$. Similarly, the "global store" monad can be specified by two operations *lookup* and *update*, and seven equational laws [7]. Many other computational aspects can be specified in this way; see e.g. [6].

The specification (T, Σ, Γ) is encoded in Coq as a module signature, i.e., `Module Type` specializing `MONAD_INTERFACE`, like the following:

```
Module Type MAYBEMONAD_INTERFACE <: MONAD_INTERFACE.
Parameter Nothing : forall (A: Type), M A.
Axiom Strictness : forall (A B : Type) (f : A -> M B),
   (Nothing A) >>= f  =  (Nothing B).
End MAYBEMONAD_INTERFACE.
```

Then, we can start reasoning about (and implementing) programs with effects by *assuming* a monad implementing this signature. Program specifications can be given using the equational logic at the `Prop` level of Coq. For instance, the specification of a program for in-place swapping of two locations, in the monad for global store, is the following:

```
Module StateInstance <: STATEMONAD_INTERFACE.
Include STATEMONAD_INTERFACE.
Include MemoryState.

Lemma swap_locs : forall (l1 l2 : loc), l1 <> l2 -> {c : M unit |
 ((c >>= (fun _ => lookUp l2)) =
             (lookUp l1) >>= fun x => c >>= (fun _ => ret x)) /\
 ((c >>= (fun _ => lookUp l1)) =
             (lookUp l2) >>= fun x => c >>= (fun _ => ret x)) /\
 forall (l : loc), (l <> l1 /\ l <> l2) -> ((c >>= fun _ => lookUp(l))) =
             ((lookUp(l) >>= fun x => c >>= fun _ => ret x ))}.
```

This kind of `Lemma`ta can be proved constructively as usual, by providing a program `c` and proving that it meets the specification. This proof will make use of the abstract algebraic laws declared in the monad signature (`STATEMONAD_INTERFACE` in this case). Notice that there is no need to provide any real implementation of the monadic specification in Coq, in order to program with the operators and prove the specification. Actually, it is not even advisable: for proving that programs are compliant to their specifications we cannot rely on peculiar properties of any specific implementation.

At this point, from these proofs we can extract Haskell programs by taking advantage of the standard Coq `Extraction` facility. The programs so obtained cannot be executed, because they will contain the constructors op_i which have still to be defined. Differently from previous work [3], we solve this issue by automatically replacing during the `Extraction` each op_i with a suitable Haskell code fragment, possibly using operators of the corresponding Haskell monad. In the case of the "maybe" monad above, we declare:

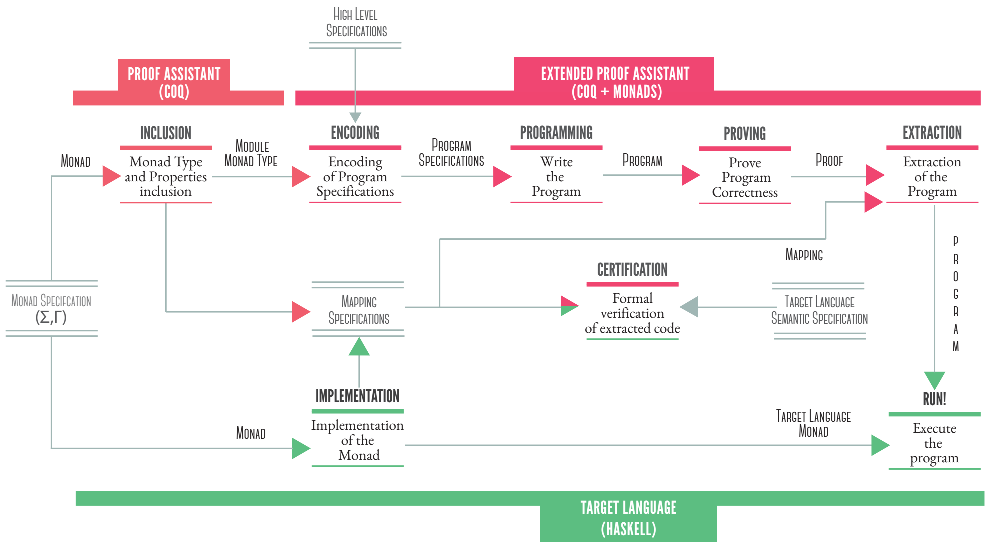
```
Extract Constant Maybe.M "a" => "Maybe a".
Extract Constant Maybe.ret => "Just".
Extract Constant Maybe.bind => "(>>=)".
Extract Constant Maybe.Nothing => "Nothing".
```

This is the step where we provide the implementation of the monad; in general, each operator can be mapped to an arbitrary complex code snippet. With these definitions, the extracted code can be readily executed in the Haskell runtime, with the proper monads covering the non-functional computational aspects.

Still, we have to prove that the mappings defined in the `Extract Constant` declarations are sound. This corresponds to prove that the equational laws declared in the monad interface are respected. The methodology for proving this soundness is general and uniform, and proceeds as follows. Let $s = t$ be an equational law of the monad specification, and let s', t' the two Haskell programs obtained by extraction from s, t, respectively; we have to prove that s' and t' are *semantically equivalent* with respect to the semantics of Haskell.

Now, instead of working with the full-blown Haskell syntax and semantics, it is more convenient to work with the *Core language*, a very small, explicitly-typed, variant of System F used as an intermediate language in `ghc`. On this language we can easily define an *applicative bisimulation* $M \approx N$ à la Abramsky [1], which corresponds to the behavioural (i.e., contextual) equivalence. Thus, for each pair s', t' as above, let s'', t'' be the corresponding two Core terms produced by `ghc` (with suitable options); we have to prove that $s'' \approx t''$. Once all equivalences $s'' \approx t''$ have been proved, we can assert that the mapping defined by the `Extract Constant` clauses is correct with respect to the equational laws assumed in the monad signature, and hence the extracted code with effects is certified.

These equivalences can be proved "on the paper", following the usual techniques for applicative bisimulation. However, these proofs can be quite long and error-prone, hence it is better (and safer) to develop them within a proof assistant. To support these proofs we are currently developing a formalization in Coq of the syntax, semantics and behavioural equivalence of Core language, similar to that in [5]. The whole methodology is summarized in the following diagram.


High Level Specifications
Proof Assistant (Coq)
Extended Proof Assistant (Coq + Monads)
Inclusion
Monad Type and Properties inclusion
Module Monad Type
Encoding
Encoding of Program Specifications
Program Specifications
Programming
Write the Program
Program
Proving
Prove Program Correctness
Proof
Extraction
Extraction of the Program
Monad
Mapping
Program
Monad Specifcation (Σ,Γ)
Mapping Specifications
Certification
Formal verification of extracted code
Target Language Semantic Specification
Implementation
Implementation of the Monad
Monad
Target Language Monad
Run!
Execute the program
Target Language (Haskell)

References

[1] S. Abramsky. The lazy lambda calculus. *Research topics in functional programming*, pages 65–116, 1990.

[2] P. Letouzey. Extraction in Coq: An overview. In *Proc. CiE*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

[3] M. Miculan and M. Paviotti. Synthesis of distributed mobile programs using monadic types in Coq. In *Proc. ITP'12*, LNCS 7406. Springer, 2012.

[4] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[5] M. Pirog and D. Biernacki. A systematic derivation of the STG machine verified in Coq. In *ACM Sigplan Notices*, volume 45, pages 25–36. ACM, 2010.

[6] G. D. Plotkin and A. J. Power. Computational effects and operations: An overview. *Electr. Notes Theor. Comput. Sci.*, 73:149–163, 2004.

[7] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

Toward a Theory of Contexts of Assumptions in Logical Frameworks

Amy Felty[1], Alberto Momigliano[2], and Brigitte Pientka[3]

[1] School of Electrical Engineering and Computer Science, University of Ottawa, Canada, afelty@eecs.uottawa.ca
[2] Dipartimento di Informatica, Università degli Studi di Milano, Italy, momigliano@di.unimi.it
[3] School of Computer Science, McGill University, Montreal, Canada, bpientka@cs.mcgill.ca

In the beginning Gentzen created natural deduction, but then He switched to the sequent calculus in order to sort out the meta-theory. Something similar happened to logical frameworks supporting higher-order abstract syntax (HOAS): first Edinburgh LF adopted Martin-Löf's parametric-hypothetical judgments to encode object logics in such a way that contexts were left *implicit*. Later on, Twelf [5] had to provide some characterization of contexts (regular worlds) to verify the meta-theory of those very object logics. The same applies to λProlog vs. Abella [3] and Hybrid [2] and, in a more principled way, to Beluga [4].

One may argue that, prior to Girard, proof-theory had been somewhat oblivious to what contexts look like. Even sub-structural logics view a context of assumptions as a *flat* collection of formulas $A_1, A_2, \ldots, A_n$ listing its elements separated by commas. However, this turns out to be inadequate once we mechanize this matter, as it ignores that assumptions come in *blocks*. Consider as an object logic the typing rules for the polymorphic lambda-calculus:

$$\frac{}{x\ \mathsf{term}}\ tm_x \qquad \frac{}{x : T_1}\ of_v \qquad\qquad \frac{}{\alpha\ \mathsf{tp}}\ tp_v$$

$$\frac{\begin{array}{c}\vdots\\ M : T_2\end{array}}{(\mathsf{lam}\, x.\, M) : (\mathsf{arr}\, T_1 T_2)}\ of_l^{tm_x, of_v} \qquad\qquad \frac{\begin{array}{c}\vdots\\ M : T_1\end{array}}{(\mathsf{tlam}\, \alpha.\, M) : (\mathsf{all}\, \alpha.\, T)}\ of_{tl}^{tp_v}$$

$$\frac{M_1 : (\mathsf{arr}\, T_1 T_2) \quad M_2 : T_1}{(\mathsf{app}\ M_1\ M_2) : T_2}\ of_a \qquad\qquad \frac{M : (\mathsf{all}\, \alpha.\, T_1) \quad T_2\ \mathsf{tp}}{(\mathsf{tapp}\ M\ T_1) : [T_2/\alpha]T_1}\ of_{ta}$$

We have proposed in [1] to view contexts as *structured sequences* of declarations D, where a declaration is a block of *unique* (atomic) assumptions separated by ';'.

$$\begin{array}{rcll}
\text{Atom} & A & & \\
\text{Block of declarations} & D & ::= & A \mid D; A \\
\text{Context} & \Gamma & ::= & \cdot \mid \Gamma, D \\
\text{Schema} & S & ::= & D_s \mid D_s + S
\end{array}$$

A schema classifies contexts and consists of declarations D_s, possibly more general than those occurring in a concrete context having schema S. This yields for the above example

$$\begin{array}{rcl}
\Gamma & ::= & \cdot \mid \Gamma, (x\ \mathsf{term}; x : T) \mid \Gamma, \alpha\ \mathsf{tp} \\
S & ::= & \alpha\ \mathsf{tp} + (x\ \mathsf{term}; x : T)
\end{array}$$

where, e.g., the context $\alpha_1\ \mathsf{tp}$, $(x_1\ \mathsf{term}; x_1 : (\mathsf{arr}\ \ \alpha_1\ \alpha_1))$, $(x_2\ \mathsf{term}; x_2 : \alpha_1)$ has schema S.

Since contexts are structured sequences, they admit structural *properties* on the level of sequences (for example by adding a new declaration) as well as inside a block of declarations (for example by adding an element to an existing declaration). We distinguish also between structural properties of a *concrete* context and structural properties of *all* contexts of a given schema.

We give a unified treatment of all such weakening/strengthening/exchange re-arrangements via total operations rm and perm that *remove* an element of a declaration, and *permute* elements within a declaration. For example, declaration weakening can be seen as:

$$\frac{\Gamma, \mathsf{rm}_A(D), \Gamma' \vdash J}{\Gamma, D, \Gamma' \vdash J}\ \textit{d-wk}$$

Suppose now that we want to prove in a logical framework some meta-theorem involving different contexts, say "if $\Gamma_1 \vdash J_1$ then $\Gamma_2 \vdash J_2$", for Γ_i of schema S_i. HOAS-based logical frameworks have so far pursued two apparently different options:

(G) We reinterpret the statement in a *generalized context* containing all the relevant assumptions—we call this the *generalized context* approach, as taken in Twelf and Beluga—and prove "if $\Gamma_1 \cup \Gamma_2 \vdash J_1$ then $\Gamma_1 \cup \Gamma_2 \vdash J_2$", where "$\cup$" denotes the *join* of the two contexts.

(R) We state how two (or more) contexts are *related*—we call this the *context relations* approach. The statement becomes therefore "if $\Gamma_1 \sim \Gamma_2$ and $\Gamma_1 \vdash J_1$ then $\Gamma_2 \vdash J_2$", with an explicit and typically inductive definition of this relation. This approach is taken in Abella and Hybrid.

If we had a common grounding of both approaches, this would pave the way toward moving proofs from one system to another, in particular breaking the type/proof theory barrier. It turns out, roughly, that a context relation can be seen as the graph of one or more appropriate rm operation on a generalized context. Further, if we take the above join metaphor seriously, we can organize declarations and contexts in a *semi-lattice*, where $x \preceq y$ holds iff x can be reached from y by some rm operation on y. A generalized context will indeed be the (lattice-theoretic) join of two contexts and context relations can be identified by navigating the lattice starting from the join of the to-be-related contexts. Our ongoing effort is to use the lattice structure to give a declarative account *promotion/demotion* of theorems (known in the Twelf lingo as "context subsumption"), where a statement proven in a certain context can be used in a "related" one. We may formulate subsumption rules akin to upward and downward casting over the lattice order.

This work also has a practical outcome in our ongoing work designing *ORBI* (Open challenge problem Repository for systems supporting reasoning with BInders), a repository for sharing benchmark problems and their solutions for HOAS-based systems, in the spirit of TPTP [6].

References

[1] A. Felty, A. Momigliano, and B. Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1—a foundational view. Submitted, 2014.

[2] A. P. Felty and A. Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

[3] A. Gacek. The Abella interactive theorem prover (system description). In *IJCAR 2008*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.

[4] B. Pientka and J. Dunfield. Beluga:a framework for programming and reasoning with deductive systems (system description). In *IJCAR 2010*, volume 6173 of *LNCS*, pages 15–21. Springer, 2010.

[5] C. Schürmann. The Twelf proof assistant. In *22nd International Conference on Theorem Proving in Higher Order Logics*, volume 567 of *LNCS*, pages 79–83. Springer, 2009.

[6] G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

Modular and lightweight certification of polyhedral abstract domains*

Alexis Fouilhe, Sylvain Boulmé, and Michaël Périn

Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
{alexis.fouilhe,sylvain.boulme,michael.perin}@imag.fr

Abstract interpretation [5] provides a theory for static analysis of programs, where sets of reachable states are over-approximated by elements of an *abstract domain*. In particular, the domain of *convex polyhedra* [6] expresses postconditions as conjunctions of affine inequalities: a polyhedron p encodes a formula "$\bigwedge_i \sum_j a_{ij}.x_j \leq b_i$", where a_{ij} and b_i are rational constants, and x_j are numerical variables of the program. Its semantics (or concretization) is the *predicate* $[\![p]\!]$ defined as "$\lambda m. \bigwedge_i \sum_j a_{ij}.m(x_j) \leq b_i$", where m is a total map from variables to rationals representing a *memory state*. The analyzer computes postconditions in a given abstract domain by performing a symbolic evaluation of programs that combines *operators* of this domain. Its correctness relies on each domain operator over-approximating a given *predicate transformer*.

An abstract interpreter such as ASTRÉE [3] is able to ensure the absence of undefined behaviours in large critical programs from avionics. But ASTRÉE is itself very complex and, despite the care put in its development, it may contain bugs. This is probably also the case for well-known abstract domain implementations, such as the PPL [1] and APRON [8]. Inspired by the development in COQ of the COMPCERT certified compiler [10], the VERASCO project aims to build a certified abstract interpreter [4]. Our work in this project focuses on obtaining a provably correct library for convex polyhedra, similar in features and performance to the core of the PPL and APRON polyhedra libraries.

Proving correct the result of domain operators on polyhedra reduces to proving inclusions of polyhedra: a polyhedron p is included in a polyhedron p' iff $\forall m, [\![p]\!](m) \Rightarrow [\![p']\!](m)$. If each inequality of p' is entailed by a positive linear combination of the inequalities of p, then inclusion holds. Farkas's lemma states that when inclusion holds, such a vector Λ of linear combinations always exists.

Moreover, such a Λ can be considered as a certificate containing the necessary information to build the result p' of a given domain operator from its operands which are here expressed as p. The result $p' = \Lambda.p$ satisfies the inclusion properties which guarantee its correctness, by construction. Our certified abstract domain of polyhedra is built out of two components:

- An untrusted OCAML backend which, for each operator, produces certificates.
- A frontend, developed in COQ, which builds proved-correct results using certificates provided by the backend.

This idea has previously been experimented by Frédéric Besson et al. [2]. Our work makes the frontend more modular and more generic with respect to the backend. All that is required from the backend is to be able to generate certificates in our format. The backend could use its own data structures (e.g. double representation), or trade some precision for computationally cheaper operators [11, 9]. Such flexibility is achieved reducing the coupling between the frontend and the backend:

- Communication between the frontend and the backend is reduced to certificates, i.e. descriptions of linear combinations of inequalities identified by integers.

*This work was partially supported by ANR project "VERASCO" (INS 2011).

- The frontend ensures soundness but does not give formal precision guarantees. The precision versus efficiency trade-off is delegated to the backend.

The frontend requires the backend to implement only a basic OCAML interface. It is extended in the frontend using functors: extra features are added in a modular way to any numerical domain without relying on its specifics. For example, the predicate transformer for assignment can be phrased in terms of more basic operators: intersection, projection and renaming. A functor adds the operator to a basic domain and builds the required correctness proofs.

To complete our abstract domain, we built a backend using a constraints-only representation of polyhedra [7]. Its operators use tweaked versions of standard algorithms so as to produce certificates as a cheap by-product of computations. Experiments show that the overhead of result verification is sufficiently low for our abstract domain to remain competitive with well-established, but non-verifying, implementations.

In conclusion, result verification is particularly well suited for certifying polyhedral abstract domains. Our work demonstrates an efficient, evolutive and reusable design, which could serve as a guiding example for lightweight certification. We hope to extend this work to a whole static analyzer.

References

[1] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2), 2008.

[2] F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Technical Report RR-6333, INRIA, 2007.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*. ACM, 2003.

[4] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal Verification of a C Value Analysis Based on Abstract Interpretation. In *SAS*, volume 7935 of *LNCS*. Springer, 2013.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM, 1977.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*. ACM, 1978.

[7] A. Fouilhe, D. Monniaux, and M. Périn. Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In *SAS*, volume 7935. Springer, 2013.

[8] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.

[9] V. Laviron and F. Logozzo. Subpolyhedra: a (more) scalable approach to infer linear inequalities. In *VMCAI*, volume 5403 of *LNCS*, pages 229–244. Springer, 2009.

[10] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[11] S. Sankaranarayanan, M. Colón, H. Sipma, and S. Manna. Efficient strongly relational polyhedral analysis. In *VMCAI*, volume 3855 of *LNCS*, pages 111–125. Springer, 2006.

The Church-Scott representation of inductive and coinductive data in typed lambda calculus

Herman Geuvers[1,2]

[1] ICIS, Radboud University Nijmegen, the Netherlands
[2] Technical University Eindhoven, the Netherlands

Data in the lambda calculus is usually represented using the "Church encoding", which gives closed terms for the constructors and which naturally allows to define functions by "iteration". An additional nice feature is that in system F (polymorphically typed lambda calculus) one can define closed data types for this data, the iteration scheme is well-typed and beta-reduction is always terminating. A problem is that primitive recursion is not directly available: it can be coded in terms of iteration at the cost of inefficiency (e.g. a predecessor with linear run-time). The much less well-known Scott encoding [1] has case distinction as a primitive. (For the numerals, these are also known as 'Parigot numerals'[3,4].) The terms are not typable in system F and there is no iteration scheme, but there is a constant time destructor (e.g. predecessor).

We will present a unification of the Church and Scott presentation of data types, which has primitive recursion as basic. We show how these can be typed in the polymorphic lambda calculus extended with recursive types and we show that all terms are strongly normalizing. We also show that this works for the dual case, co-inductive data types, and we show how programs can be extracted from proofs in second order predicate logic.

Church and Scott data types As a step back, we look at data in the untyped λ-calculus. Church numerals:

$$\begin{array}{rcl} \overline{0} & := & \lambda x\, f.x \\ \overline{1} & := & \lambda x\, f.f\,x \\ \overline{2} & := & \lambda x\, f.f\,(f\,x) \end{array} \qquad \begin{array}{rcl} \overline{p} & := & \lambda x\, f.f^p\,(x) \\ \overline{S} & := & \lambda n.\lambda x\, f.f\,(n\,x\,f) \end{array}$$

The Church numerals have *iteration* as basis: the numerals are iterators. An advantage is that one gets quite a bit of "well-founded recursion" for free. A disadvantage is that there is no pattern matching built in, so the predecessor is hard to define.

Scott numerals:

$$\begin{array}{rcl} \underline{0} & := & \lambda x\, f.x \\ \underline{1} & := & \lambda x\, f.f\,\underline{0} \\ \underline{2} & := & \lambda x\, f.f\,\underline{1} \end{array} \qquad \begin{array}{rcl} \underline{p+1} & := & \lambda x\, f.f\,\underline{p} \\ \underline{S} & := & \lambda n.\lambda x\, f.f\,n \end{array}$$

The Scott numerals have *case* as a basis: the numerals are *case distinctors*: $\underline{n}AB = A$ if $n = 0$ and $\underline{n}AB = B\underline{m}$ if $n = m + 1$. An advantage is that the predecessor can immediately be defined, but one has to get "recursion" from somewhere else (e.g. by using a fixed point-combinator).

A more general definition of Church and Scott data in the untyped λ-calculus is the following. Given a data type with constructors $\mathbf{c}_1, \ldots, \mathbf{c}_k$, each with a fixed arity, say the arity of constructor $\mathbf{c}_i$ is $\mathsf{ar}(i)$, we have a *Church encoding*:

$$\overline{\mathbf{c}_i} \;:=\; \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\lambda c_1 \ldots c_k.c_i\,(x_1\,\vec{c}) \ldots (x_{\mathsf{ar}(i)}\,\vec{c}) \qquad \begin{array}{rcl} \overline{0} & := & \lambda x\, f.x \\ \overline{S} & := & \lambda n.\lambda x\, f.f\,(n\,x\,f) \end{array}$$

and a *Scott encoding*:

$$\underline{\mathbf{c}_i} \;:=\; \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\lambda c_1 \ldots c_k.c_i\,x_1 \ldots x_{\mathsf{ar}(i)} \qquad \begin{array}{rcl} \underline{0} & := & \lambda x\, f.x \\ \underline{S} & := & \lambda n.\lambda x\, f.f\,n \end{array}$$

The Scott encoding is simpler, but it's not very well-known and seldom used. Why? Probably the main reason is that Church data can be typed in the polymorphic λ-calculus $\lambda 2$, and already quite a lot of functions can be typed in simple type theory, $\lambda\!\rightarrow$.

To type Church numerals of type $\mathtt{nat}$, we need $\mathtt{nat} = A \to (A \to A) \to A$. (Church 1940). In polymorphic λ-calculus, we can even do better by taking

$$\mathtt{nat} := \forall X.X \to (X \to X) \to X.$$

There is a (well-known) general pattern behind this

$$\begin{array}{rcl} \mathtt{bool} & := & \forall X.X \to X \to X \\ \mathtt{list}_A & := & \forall X.X \to (A \to X \to X) \to X \\ \mathtt{bintree}_{A,b} & := & \forall X.(A \to X) \to (B \to X \to X \to X) \to X \end{array}$$

This provides a nice *function definition scheme* in $\lambda 2$. As an example we give the *iteration scheme* for $\mathtt{nat}$ and $\mathtt{list}_A$. (Let D be a type.)

$$\frac{d : D \quad f : D \to D}{\mathsf{It}\, d\, f : \mathtt{nat} \to D} \qquad\qquad \frac{d : D \quad f : A \to D \to D}{\mathsf{It}\, d\, f : \mathtt{list}_A \to D}$$

with
$\mathsf{It}\, d\, f\, \overline{0} \twoheadrightarrow d$
$\mathsf{It}\, d\, f\, (\overline{S}\, x) \twoheadrightarrow f\, (\mathsf{It}\, d\, f\, x)$

with
$\mathsf{It}\, d\, f \overline{\mathtt{nil}} \twoheadrightarrow d$
$\mathsf{It}\, d\, f\, (\overline{\mathtt{cons}}\, a\, x) \twoheadrightarrow f\, a\, (\mathsf{It}\, d\, f\, x)$

Important features: (1) This scheme is available in $\lambda 2$: we can define It for $\mathtt{nat}$, $\mathtt{list}_A$, ...; (2) Using this we can define as λ-terms very many functions: $+$, $\times$, exp, Ackermann, ..., map-list, fold, ...; (3) Because these terms are typed in $\lambda 2$, these are all terminating.

Do we have types for Scott data? To type Scott numerals we need $\mathtt{nat} = A \to (\mathtt{nat} \to A) \to A$.

In $\lambda 2$, we cannot do this, unless we extend it with (positive) recursive types, obtaining $\lambda 2\mu$:

$$\mathtt{nat} := \mu Y.\forall X.X \to (Y \to X) \to X.$$

Type formation rule: $\mu Y.\Phi[Y]$ is a well-formed type if Y occurs positive in the type expression $\Phi[Y]$. A drawback of this approach is that one does not get any well-founded recursion "for free".

Combined Church-Scott encoding To get the best of both worlds, we can define the CS (Church-Scott) numerals by

$$\begin{array}{rcl} \underline{\overline{0}} & := & \lambda x\, f.x \\ \underline{\overline{1}} & := & \lambda x\, f.f\, \underline{0}\, (\underline{0}\, x\, f) \\ \underline{\overline{2}} & := & \lambda x\, f.f\, \underline{1}\, (\underline{1}\, x\, f) \end{array} \qquad\qquad \begin{array}{rcl} \underline{\overline{p+1}} & := & \lambda x\, f.f\, \underline{p}\, (\underline{p}\, x\, f) \\ \underline{\overline{S}} & := & \lambda n.\lambda x\, f.f\, n\, (n\, x\, f) \end{array}$$

These numerals can be types as well in $\lambda 2\mu$:

$$\mathtt{nat} := \mu Y.\forall X.X \to (Y \to X \to X) \to X.$$

The advantage is that now one obtains the primitive *recursion scheme* for free.

$$\frac{d : D \quad f : \mathtt{nat} \to D \to D}{\mathsf{Rec}\, d\, f : \mathtt{nat} \to D}$$

with $\mathsf{Rec}\, d\, f := \lambda n : \mathtt{nat}.n\, d\, f$, satisfying $\mathsf{Rec}\, d\, f\, \underline{\overline{0}} = d$ and $\mathsf{Rec}\, d\, f\, (\underline{\overline{S}}\, x) = f\, x\, (\mathsf{Rec}\, d\, f\, x)$.

For other known data types, we can do the same, if we observe that

- in $\lambda 2$, $\texttt{nat} := \text{lfp}\,\Phi$, with $\Phi(X) = 1 + X$ and $\text{lfp}\,\Phi$ is the well-known definable (weak) least fixed point in $\lambda 2$,
- in our new definition $\texttt{nat} := \mu Y.\text{lfp}\,\Phi^{\times Y}$, where $\Phi^{\times Y}(X) = 1 + (Y \times X)$.

The fact that all this works is due to that fact that we can define *recursive algebras* (in the terminology of [2]) inside $\lambda 2\mu$ in a very generic way.

The dual case: streams It is well-know that streams over a base type A can also be defined in $\lambda 2$:

$$\texttt{Str}_A := \exists X.X \times (X \to A \times X)$$

If we use the same type of implicit Curry-style typing for $\exists$ that we have also used for $\forall$ above, and we use $\langle -, - \rangle$ for pairing and $(-)_1$ and $(-)_2$ for the projections. we see that the definitions of head and tail are:

$$\begin{array}{rcl} \texttt{hd}\,s & := & (s_2\,s_1)_1 \\ \texttt{tl}\,s & := & \langle (s_2\,s_1)_2, s_2 \rangle \end{array}$$

Now it is hard to define the cons operator, that takes an $a : A$ and an $s : \texttt{Str}_A$ to create $\texttt{cons}\,a\,s : \texttt{Str}_A$. However, in $\lambda 2\mu$ we can define a *co-recursive co-algebra* for the functor $X \mapsto A \times X$ as follows:

$$\texttt{Str}_A := \mu Y.\exists X.X \times (X \to A \times (X + Y))$$

Now we can define

$$\begin{array}{rcl} \texttt{hd}\,s & := & (s_2\,s_1)_1 \\ \texttt{tl}\,s & := & \text{case}\,(s_2\,s_1)_2 \text{ of } (\texttt{inl}x \Rightarrow \langle x, s_2 \rangle)\ (\texttt{inr}y \Rightarrow y) \\ \texttt{cons}\,a\,s & := & \langle a, \lambda x.\langle a, \texttt{inr}\,s \rangle \rangle \end{array}$$

And we can check that

$$\begin{array}{rcl} \texttt{hd}(\texttt{cons}\,a\,s) & := & a \\ \texttt{tl}(\texttt{cons}\,a\,s) & := & s \end{array}$$

It can be shown that the approach sketched above works for more general inductive and co-inductive data-types. It gives “nice” finite representations of infinite data in the untyped lambda calculus. (E.g. the stream of natural numbers is a term in normal form.) It can also be shown that the programs from proof extraction mechanism as developed by Krivine, Leivant and Parigot [3] works nicely for these data type definitions.

Reference

[1] M. Abadi, L. Cardelli and G. Plotkin, Types for the Scott Numerals, 1993, `http://lucacardelli.name/Papers/Notes/scott2.pdf`.

[2] H. Geuvers, Inductive and Coinductive Types with Iteration and Recursion, in the informal proceedings of the 1992 workshop on Types for Proofs and Programs, Bastad 1992, Sweden, eds. B. Nordström, K. Petersson and G. Plotkin, pp 183–207. `http://www.cs.ru.nl/~herman/PUBS/BRABasInf_RecTyp.ps.gz`

[3] P. Fu, A. Stump, Self Types for Dependently Typed Lambda Encodings, Submitted to RTA-TLCA 2014.

[4] M. Parigot, Recursive Programming with Proofs, Theor. Comput. Sci., 94, 2, 1992, pp. 335-336.

A type system for Continuation Calculus

Herman Geuvers[1,2], Wouter Geraedts[1], Bram Geron[3], Judith van Stegeren[1]

[1] Radboud University Nijmegen, the Netherlands
[2] Technical University Eindhoven, the Netherlands
[3] School of Computer Science, University of Birmingham, UK

Abstract

Continuation Calculus (CC), introduced by Geron and Geuvers [2], is a simple foundational model for functional computation. It is closely related to lambda calculus and term rewriting, but it has no variable binding and no pattern matching. It is Turing complete and evaluation is deterministic. Notions like "call-by-value" and "call-by-name" computation are available by choosing appropriate function definitions: e.g. there is a call-by-value and a call-by-name addition function.

In the present paper we extend CC with types, to be able to define data types in a canonical way, and functions over these data types, defined by iteration. Data type definitions follow the so-called "Scott encoding" of data, as opposed to the more familiar "Church encoding".

The iteration scheme comes in two flavors: a call-by-value and a call-by-name iteration scheme. The call-by-value variant is a double negation variant of call-by-name iteration. The double negation translation allows to move between call-by-name and call-by-value.

Continuation calculus (or CC) [2] is a crossover between term rewriting systems and λ-calculus. Rather than focusing on expressions, continuation calculus treats continuations as its fundamental object. This is realized by restricting evaluation to strictly top-level, discarding the need for evaluation inside contexts. This also fixes an evaluation order, so the representation of a program in CC depends on whether call-by-value or call-by-name is desired. Furthermore, CC "separates code from data" by placing the former in a static *program*, which is sourced for reductions on a term. Variables are absent from terms, and no substitution happens inside terms.

Despite the obvious differences between CC and λ-calculus with continuations (or λ_C), there seems to be a strong correspondence. For instance, it has been suggested [3] that programs in either can be simulated in the other up to parametrized (non)termination, in an untyped setting. If the correspondence turns out to be sufficiently strong, continuation calculus could become an alternative characterization of λ_C, and theorems in one system could apply without much effort to the other.

The purpose of this paper is to strengthen the correspondence between CC and λ-calculus, by introducing a type system for CC and by showing how data types and functions over data can be defined in CC. The type system rejects some undesired terms and the types emphasize the difference between call-by-name and call-by-value. Also, the types pave the way for proving properties of the programs. The types themselves do not enforce termination, because the system is 'open': one can add whatever program one wants. However, if the programs on data types are defined using only iteration and non-circular program rules, all programs are terminating, which we show using a translation to a simply typed λ-calculus with data-types and iterators in call-by-name and call-by-value style. We show this λ-calculus to be strongly normalizing.

Informal definition of CC Terms in CC are of the shape $n.t_1.t_2.\ldots.t_k$, where n is a name and t_i is again a term. The 'dot' denotes binary application, which is left-associative. In CC, terms can be evaluated by applying *program rules* which are of the shape

$$n.x_1.x_2.\ldots.x_p \longrightarrow u, (*)$$

where u is a term over variables $x_1 \ldots x_p$. However, this rule can only be applied on the 'top level':

- reduction is not a congruence;
- rule (*) can only be applied to the term $n.t_1.t_2.\dots.t_k$ in case $k = p$,
- then this term evaluates to $u[t_1/x_1, \dots, t_p/x_p]$.

CC has no pattern matching or variable binding, but it is Turing complete and a faithful translation to and from the untyped λ-calculus can be defined, see [3].

In continuation calculus, the natural numbers are represented by the names **Zero** and **Succ** and the following two program-rules:

$$\begin{array}{lcl} \mathbf{Zero}.c_1.c_2 & \longrightarrow & c_1 \\ \mathbf{Succ}.x.c_1.c_2 & \longrightarrow & c_2.x \end{array}$$

So **Zero** represents 0, **Succ**.**Zero** represents 1, **Succ**.(**Succ**.**Zero**) represents 2 etcetera. This representation of data follows the so-called Scott encoding, which is known from the untyped lambda calculus by defining **Zero** $:= \lambda x\,y.x$, **Succ** $:= \lambda n.\lambda x\,y.y\,n$ (e.g. see [1, 4]). The Scott numerals have "case-distinction" built in (distinguishing between 0 and $n + 1$), which can be used to mimic pattern matching. The more familiar Church numerals have iteration built in. For Scott numerals, iteration has to be added, or it can be obtained from the fixed-point combinator in the case of untyped lambda calculus. For CC the situation is similar: we have to add iteration ourselves.

As an example, we define addition in two ways: in call-by-value (CBV) and in call-by-name (CBN) style ([2]).
Example 1.

$$\begin{array}{lcl} \mathbf{AddCBV}.n.m.c & \longrightarrow & n.(c.m).(\mathbf{AddCBV}'.m.c) \\ \mathbf{AddCBV}'.m.c.n' & \longrightarrow & \mathbf{AddCBV}.n'.(\mathbf{Succ}.m).c \end{array}$$

$$\begin{array}{lcl} \mathbf{AddCBN}.n.m.c_1.c_2 & \longrightarrow & n.(m.c_1.c_2).(\mathbf{AddCBN}'.m.c_2) \\ \mathbf{AddCBN}'.m.c_2.n' & \longrightarrow & c_2.(\mathbf{AddCBN}.n'.m) \end{array}$$

For **AddCBV** *we find that* **AddCBV**.(**Succ**^{n}.**Zero**).(**Succ**^{m}.**Zero**).c *evaluates to* c.(**Succ**$^{n+m}$.**Zero**)*: the result of the addition function is computed completely and passed as argument to the continuation* c*. For* **AddCBN***, only a first step in the computation is carried out and then the result is passed to the appropriate continuation* c_1 *or* c_2*.*

Continuation calculus as it occurs in [2] is untyped. In the present work we present a typing system for continuation calculus, using simple types and positive recursive types. The typing system gives the user some guarantee about the meaning and well-formedness of well-typed terms. We also develop a general procedure for defining algebraic data-types as types in CC and for transforming functions defined over these data types into valid typed terms in CC.

References

[1] M. Abadi, L. Cardelli, and G. Plotkin. Types for the scott numerals. `http://lucacardelli.name/Papers/Notes/scott2.pdf`, 1993.

[2] B. Geron and H. Geuvers. Continuation calculus. In *Proceedings of COS 2013*, volume 127 of *EPTCS*, pages 66–85, 2013.

[3] Bram Geron. Continuation calculus, master's thesis. `http://alexandria.tue.nl/extra1/afstversl/wsk-i/geron2013.pdf`, 2013.

[4] J.M. Jansen. Programming in the λ-calculus: From Church to Scott and back. In *The Beauty of Functional Code*, volume 8106 of *Lecture Notes in Computer Science*, pages 168–180, 2013.

Church-Rosser Theorem for sequent lambda calculi*

Silvia Ghilezan[1], Jelena Ivetić[1], and Silvia Likavec[2]

[1] University of Novi Sad, Faculty of Technical Sciences, Serbia
`gsilvia, jelenaivetic@uns.ac.rs`
[2] Dipartimento di Informatica, Università di Torino, Italy
`likavec@di.unito.it`

The Curry-Howard correspondence is a well-known relation between typed lambda calculi and intuitionistic natural deduction. In the last decade, several term calculi have been designed to embody the Curry-Howard correspondence for intuitionistic sequent calculus, [5, 2, 3, 4] among others.

Our focus is on the untyped version of the so called λ^{Gtz}-calculus of [3], which is known to be non-confluent. We regain confluence in the untyped λ^{Gtz}-calculus by imposing restrictions on the reduction rules, which eliminate the critical pair. In this way two subcalculi, the call-by-name λ_N^{Gtz} and the call-by-value λ_V^{Gtz} variant of λ^{Gtz} are obtained. In the $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [1], the call-by-name and call-by-value fragments are obtained by dual restrictions on the reductions rule, here instead the situation is different in the call-by-name case. The set of λ_V^{Gtz}-terms is equal to the set of λ^{Gtz}-terms and the the call-by-value λ_V^{Gtz} subcalculus is obtained by restrictions on the reductions. However, the set of λ_N^{Gtz}-terms is a strict subset of the set of λ^{Gtz}-terms. Therefore, we introduce an appropriate mapping to translate λ-terms into λ_N^{Gtz}-terms. We prove then that this mapping preserves the operational semantics of lambda calculus as well as the normal forms.

We prove the confluence of the two proposed subcalculi by adapting Takahashi's parallel reductions technique, used in [6] for proving the confluence of λ-calculus. This is a refinement of the standard Tait and Martin-Löf's proof of the confluence of $\beta\eta$-reduction in the λ-calculus. We analyse the granularity of reduction rules and then define a new notion of parallel reductions in this framework. We then prove the diamond property, which yields the proof of confluence for type free λ_V^{Gtz}-calculus. Finally, we show that the diamond property of the new parallel reduction is also applicable to λ_N^{Gtz}. Confluence of a sequent lambda calculus is usually proved by embedding it in a calculus already known to be confluent. We developed a direct proof of confluence of two subcalculi of λ^{Gtz}, which was our motivation. To the best of our knowledge this is a first direct proof of confluence in sequent lambda calculi.

References

[1] P.-L. Curien and H. Herbelin. The duality of computation. In *5th International Conference on Functional Programming, ICFP'00*, pages 233–243. ACM Press, 2000.

[2] R. Dyckhoff and L. Pinto. Cut-elimination and a permutation-free sequent calculus for intuitionistic logic. *Studia Logica*, 60(1):107–118, 1998.

[3] J. Espírito Santo. Completing Herbelin's programme. In *The 8th International Conference on Typed Lambda Calculi and Applications, TLCA '07*, volume 4583 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2007.

*This work has been partly supported by the Ministry of Education and Science of Republic of Serbia, projects ON174026 and III44006.

[4] J. Espírito Santo, R. Matthes, and L. Pinto. Continuation-passing style and strong normalisation for intuitionistic sequent calculi. In *The 9th International Conference on Typed Lambda Calculi and Applications, TLCA '07*, volume 4583 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2007.

[5] H. Herbelin. A lambda calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Computer Science Logic, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1995.

[6] M. Takahashi. Parallel reduction in λ-calculus. *Information and Computation*, 118:120–127, 1995.

Session Types, Solos, and the Computational Contents of Sequent Calculus Proofs

Nicolas Guenot

IT University of Copenhagen
ngue@itu.dk

The original form of the Curry-Howard correspondence established a connection between intuitionistic natural deduction and the simply-typed λ-calculus, but this has been extended to various other logical systems and other calculi. One problem with such extensions arise when considering proofs in the sequent calculus: the correspondence in such a setting is difficult to design, so that the proof systems considered are often quite constrained — for example using a stoup in the logic to restrict the shape of proofs — and term calculi are not necessarily well-behaved — for example when one interprets the left rule for implication as a floating `let` construct, which is problematic in the usual theory of λ-calculi. We discuss in this work the status of this problem in the light of some recent developments relating linear sequent proofs to session-typed processes.

1 Cut Elimination and Permutations

A quite striking difference between normalisation in the natural deduction NJ and any cut elimination procedure for the sequent calculus is the fact that eliminating cuts from a proof is performed in small steps, such that important steps, the so-called *principal cases*, require some form of *synchronisation* of the shape of the two subproofs above the cut. The other cases are mere *trivial permutations* that reflect the lax structure of proofs in the sequent calculus: the order between the rule instances in a given proof are often irrelevant, leading to the notion of proof-nets in linear logic [Gir87].

While the view of principal cases in cut elimination as synchronisation has lead to the *concurrent* interpretation of sequent calculi, as done in linear logic [CP10], this emphasis on trivial permutations leads to the idea that instead of using proof-nets to represent parallel processes, one can adopt a syntactic approach where permutations correspond to the equations of a congruence on processes — as usually considered in the process calculi community. The correspondence we are looking for is thus:

propositions	session types
proofs	processes
cut elimination	communication
congruence	permutation

2 Linear Logic and Session Solos

The session types system introduced by Caires and Pfenning [CP10], based on the intuitionistic variant of linear logic and adapted to linear logic by Wadler [Wad12] establishes a correspondence that provides strong guarantees on typed π-terms, but the connection it shows between proofs and processes is not as tight as one might expect, in the sense that some equations on processes cannot be reflected in the structure of proofs. To improve this, we propose to use the *solos* calculus [LV03], a restricted setting where inputs and outputs are free of explicit sequentialisation — it drops the prefix operation just as the asynchronous π-calculus drops it for outputs, leaving only implicit, *causal* dependencies. Consider for example the typing of some process starting with an output in Wadler's typed π-calculus:

$$\dfrac{\overset{\Pi_1}{P \vdash \Gamma, y : A} \qquad \overset{\Pi_2}{Q \vdash \Delta, x : B}}{\overline{x}\langle y\rangle.(P \,|\, Q) \vdash \Gamma, \Delta, x : A \otimes B}$$

and observe that typing involves the decomposition of several operators, where in particular the prefix operation supposedly prevents processes inside P or Q to *move out* even if they are not using x or y. As a consequence, typeability is not preserved by the structural congruence usually applied on π-terms, as illustrated by the two terms below:

$$\overline{x}\langle y\rangle.(P \,|\, (\nu z)\,(Q \,|\, R)) \;\equiv\; \overline{x}\langle y\rangle.(\nu z)\,((P \,|\, Q) \,|\, R)$$

where the left one is typeable and the right one is not. Moving to the more parallel setting of solos allows to write this process as $(y)\,(\overline{x}\langle y\rangle \,|\, (z)\,(P|Q))$ or equivalently as $(y)\,(z)\,(\overline{x}\langle y\rangle \,|\, (P|Q))$, which is much closer to the structure of the corresponding proof in the sequent calculus, where some rule instances in Π_1 and Π_2 might permute downwards.

The adoption of solos as the underlying computational model yields a session type system where the correspondence between proofs and processes is tighter, and all this can also be adapted to the intuitionistic variant of linear logic. This might place solos in the position of being an interesting implementation language for typesafe concurrent programming, but it does not solve all problems of previous systems nor create a perfect correspondence. Because of the shape of proofs in the sequent calculus, an exact matching of π-terms or solos processes with such proofs is impossible. This offers two orthogonal possibilities: either making concessions on the side of process calculi and constrain further the structure of processes, or use a different logical formalism where the structure of proofs is even more lax than in the sequent calculus.

References

[CP10] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR'10*, volume 6269 of *LNCS*, pages 222–236, 2010.

[Gir87] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[LV03] C. Laneve and B. Victor. Solos in concert. *Mathematical Structures in Computer Science*, 13(5):657–683, 2003.

[Wad12] P. Wadler. Propositions as sessions. In P. Thiemann and R. B. Findler, editors, *ICFP'12*, pages 273–286. ACM, 2012.

Covering Spaces in Homotopy Type Theory

Kuen-Bang Hou (Favonia)[1] *

Carnegie Mellon University, USA
`favonia@cs.cmu.edu`

Homotopy type theory (HoTT) is an exciting new interpretation of intensional type theory which provides a synthetic framework for homotopy theory. It is natural to ask whether we can restate various homotopy-invariant concepts and theorems from classical homotopy theory. In this talk I will explore one fundamental construct: *covering spaces*. It turns out that we can express covering spaces (up to homotopy) elegantly in HoTT as follows.

Definition 1. A *covering space* of a type (space) A is a family of sets indexed by A.

That is, the type of a covering of A is simply $A \to \mathsf{Set}$ where Set is the type of all sets. To verify that this formulation matches the classical one, I proved in HoTT several expected properties of covering spaces, including that covering spaces of a pointed, path-connected A are classified by functors $\pi_1(A) \to \mathsf{Set}$ where $\pi_1(A)$ is the fundamental group of A, that homotopy-equivalent classes of paths with one end fixed form a universal covering space, and that simply-connectedness implies universality.

I will review the key ideas in the proofs. Some interesting techniques employed in the current proofs seem applicable to other constructions as well. It is worth emphasizing that every proof mentioned in this talk has been fully mechanized [1] in the proof assistant Agda. The code of critical parts will be demonstrated during the talk.

Looking forward, I am working on the generalization of the results from sets to groupoids. These attempts will also be discussed if time permits.

References

[1] HoTT library in Agda. https://github.com/HoTT/HoTT-Agda.

*This material is based upon work supported by the National Science Foundation under Grant No. 1116703.

Isomorphism of Finitary Inductive Types

Nicolai Kraus and Christian Sattler

University of Nottingham

Abstract

We present an algorithm for deciding isomorphism of finitary mutually inductive types.

We consider the simply typed lambda calculus with finite products and coproducts, type variables X, Y, Z, ..., and mutually inductive types, for example defined by

$$\begin{aligned} A &= Z + XA^2 + BC \\ B &= 1 + BC \\ C &= Y + AC, \end{aligned}$$

which we call (parameterized) *finitary inductive types*. Equivalently, we can define these as nested μ-expression; for example, C can be represented as

$$\mu C.Y + (\mu A.Z + XA^2 + (\mu B.1 + CB)C)C.^{1}$$

Note that the attribute *finitary* indicates the absence of function types.

Based on discussions with Swierstra and Morris, Altenkirch [1] notices that the special case of *regular* types, i.e. types defined mutually as sums over products where each product contains at most one recursive variable, corresponds to proof-relevant regular grammars where terminal symbols commute. [2] They discuss isomorphism of such types in the set model and conjectures that isomorphism of regular types is decidable, while isomorphism of general finitary inductive types might be undecidable.

In this talk, we present an algorithm that decides syntactic isomorphism of general finitary inductive types with respect to the $\beta\eta$-equational theory with strong sums and interpretation of μ-expressions as initial algebras. We further show that the set model is complete with respect to this question. This yields the solution to Altenkirch's conjectures as a special case.

We want to outline one core observation on which the easier set model variant of this algorithm is based. As is well known, in the set model parameterized finitary inductive types can equivalently be expresses as a formal power series, e.g. via the intermediate notion of finitary containers. For example, lists over X (that is $\mu A.1 + AX$) admit the representation

$$1 + X + X^2 + X^3 + X^4 + \dots, \tag{1}$$

while the series of the type C defined above starts with (we list all summands which have less than four occurrences of X, Y, Z)

$$Y + Y^2 + YZ + 3Y^3 + 3Y^2Z + YZ^2 + \dots. \tag{2}$$

It is possible that certain coefficients of the power series corresponding to a type are not finite; for example, the natural numbers $\mu A.1 + A$ have a power series over zero variables with a single

[1] We write XA^2 for $X \times A \times A$.

[2] Confusingly, other authors have introduced the term *regular functors* over type variables for the concept we name finitary inductive types here.

summand of degree 0 with coefficient of cardinality $\mathbb{N}$. This imposes additional difficulties. It forces us to treat the part consisting of the infinite coefficients separately, a finitary inductive description of which we call the *unguarded* part. However, the surgery necessary on a finitary inductive type to effectively renove the unguarded part is not canonical, leaving a residue type having only finite power series coefficients that is in general not uniquely determined.

In the set model, two types are isomorphic if the coefficients of their corresponding power series are equal. The key point now is that the guarded part is always *algebraic* over a certain function field, i.e. it is has a corresponding *minimal polynomial*, a finite representation, with coefficients in a certain kind of finitely describable algebraic structure, and two of these finite representations can be algorithmically compared. The guarded part is however selectively masked by the unguarded part, requiring further algebraic machinery involving modules, lattices, and known methods for constructively dealing with systems of polynomial equations.

Having established isomorphism in the set model between two finitary inductive types, actually synthesizing one such isomorphism as a syntactic λ-term (and thus establishing completeness of the set model) requires internalizations of parts of the above arguments into a theory not even able to reflect many basic inductive arguments. Thus, we want to stress that the actual main technical effort concerns this aspect. We strive to give our arguments in an abstract fashion, preferring type- or category-level reasoning wherever possible (e.g., making use of categorical properties of traversable functors [3]). Still, reducing the amount of low-level details in this part of our proof warrants further attention.

Our project is superficially related to previous work by Fiore [2]. Crucially, however, our recursive types are not generic, but initial: instead of just constructing and destructing elements of inductive types, we have the full power of unique existence of the folding eliminator at hand; this induces a much stronger equational theory that is more difficult to reason about.

References

[1] Thorsten Altenkirch. Isomorphisms on inductive types (talk), 2005.

[2] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 77–88, New York, NY, USA, 2004. ACM.

[3] Mauro Jaskelioff and Ondrej Rypacek. An investigation of the laws of traversals. In *MSFP*, pages 40–49, 2012.

A Separation Logic for Non-determinism and Sequence Points in C Formalized in Coq

Robbert Krebbers

ICIS, Radboud University Nijmegen, The Netherlands

Abstract

The C11 standard of the C programming language does not specify the execution order of expressions. Besides, to make more effective optimizations possible (*e.g.* delaying of side-effects and interleaving), it gives compilers in certain cases the freedom to use even more behaviors than just those of all execution orders.

Widely used C compilers exploit this freedom given by the C standard for optimizations, so it should be taken seriously in formal verification. In [3], we have presented an operational and axiomatic semantics (based on separation logic) for non-determinism and sequence points in C. Soundness of the axiomatic semantics is proved with respect to the operational semantics. This proof has been fully formalized using the Coq proof assistant.

1 Introduction

The C programming language [2] is not only among the most popular programming languages in the world, but it is also among the most dangerous programming languages. Due to weak static typing and the absence of runtime checks, it is extremely easy for C programs to have bugs that make the program crash or behave badly in other ways: NULL-pointers can be dereferenced, arrays can be accessed outside their bounds, memory can be used after it is freed, *etc.*

Instead of forcing compilers to use a predefined execution order for expressions (*e.g.* left to right), the C standard does not specify it. This is a common cause of portability and maintenance problems, as a compiler may use an arbitrary execution order for each expression. Hence, to prove the correctness of a C program with respect to an *arbitrary* compiler, one has to verify that each possible execution order is legal and gives the correct result. To make more effective optimizations possible (*e.g.* delaying of side-effects and interleaving), the C standard requires the programmer to ensure that all execution orders satisfy certain conditions. If these conditions are not met, the program may do anything. Let us take a look at an example where one of those conditions is not met.

```
int main() {
  int x; int y = (x = 3) + (x = 4);
  printf("%d %d\n", x, y);
}
```

By considering all possible execution orders, one would naively expect this program to print `4 7` or `3 7`, depending on whether the assignment `x = 3` or `x = 4` is executed first. However, the *sequence point restriction* does not allow an object to be modified more than once (or being read after being modified) between two *sequence points* [2, 6.5p2]. A sequence point occurs for example at the end `;` of a full expression, before a function call, and after the first operand of the conditional `? :` operator [2, Annex C]. Hence, both execution orders lead to a sequence point violation, and are thus illegal. As a result, the execution of this program exhibits *undefined behavior*, meaning it may do literally anything. Indeed, when compiled with `gcc -O1` (version 4.7.2), it prints `4 8`, which does not correspond to any of the execution orders.

2 Approach

As a step towards taking non-determinism and the sequence point restriction seriously in C verification, we extend our axiomatic semantics with a Hoare judgment $\{P\}\, e\, \{Q\}$ for expressions. Since expressions not only have side-effects but primarily yield a value, the postcondition Q is a function from values to assertions. Intuitively, $\{P\}\, e\, \{Q\}$ means that if P holds beforehand, and execution of e yields a value v, then $Q\, v$ holds afterwards.

Apart from partial program correctness, the judgment $\{P\}\, e\, \{Q\}$ ensures that e exhibits no undefined behavior. To deal with the unrestrained non-determinism in C, we observe that non-determinism in expressions corresponds to a form of concurrency, which separation logic is well capable of dealing with. Inspired by the rule for the parallel composition of separation logic (see [4]), we propose the following kind of rules for each operator $\circledcirc$.

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_l * P_r\}\, e_1 \circledcirc e_2\, \{Q_1 * Q_2\}}$$

The idea is that, if the memory can be split up into two disjoint parts (using the separating conjunction $*$), in which the subexpressions e_1 respectively e_2 can be executed safely, then the full expression $e_1 \circledcirc e_2$ can be executed safely in the whole memory.

To ensure no sequence point violations occur, we use separation logic with permissions [1], extended with a special class of *locked* permissions. The singleton assertion becomes $e_1 \overset{\gamma}{\mapsto} e_2$ where γ is the permission of the object e_2 at address e_1. The inference rules are set up in such a way that reads and writes are only allowed for objects that are not locked, and moreover such that objects become locked after they have been written to. At constructs that contain a sequence point, the inference rules ensure that these locks are released.

3 Formalization in Coq

All our proofs have been fully formalized using the Coq proof assistant. We used Coq's notation mechanism combined with unicode symbols and type classes for overloading to let the Coq code correspond as well as possible to the definitions on paper. Coq's type classes are also used to provide abstract interfaces for commonly used structures like finite sets and finite partial functions, so that we were able to prove theory and implement automation in an abstract way. Because the semantics is rather big, it is quite cumbersome to prove properties about it without automation, to this end, we have automated many steps of the proofs. Our Coq code, available at `http://robbertkrebbers.nl/research/ch2o`, is about 10 000 lines of code including comments and white space. Apart from that, our library on general purpose theory (finite sets, finite functions, lists, *etc.*) is about 9 000 lines.

References

[1] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, 2005.

[2] International Organization for Standardization. *ISO/IEC 9899-2011: Programming languages – C.* ISO Working Group 14, 2012.

[3] Robbert Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *POPL*, pages 101–112, 2014.

[4] Peter W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, volume 3170 of *LNCS*, pages 49–67, 2004.

All derivations of groupoid laws are propositionally equal.

Marc Lasson[1]

INRIA
PPS, Université Paris Diderot
marc.lasson@inria.fr

Garner, van den Berg [6] and Lumsdsaine [3] independently showed that in type theory, each type can be equipped with a structure of weak ω-groupoids. For this, they show that a minimal fragment MLID of Martin-Löf type theory, where identity types are the only allowed type constructors, bears a weak ω-category structure. Informally, these results state the possibility to express groupoid laws of weak ω-groupoids as types and in each case to find a canonical inhabitant of these types reflecting the fact that the law holds. Identities, inversion and concatenation of paths, associativities, idempotency of inversion, horizontal and vertical compositions of 2-paths, are all examples of groupoid laws.

In this work[1], we follow a syntactic approach proposed by Brunerie [2] to formalize the notion of groupoid laws. We call *groupoid law* any closed type $\forall \Gamma . c$ such that the sequent $\Gamma \vdash c : \mathsf{Type}$ is derivable in MLID and the context Γ is contractible. A *contractible context* is a context of the following shape: $X : \mathsf{Type}, x : X, x_1 : C_1, y_1 : M_1 = x_1, \ldots, x_n : C_n, y_n : M_n = x_n$ where x_i does not occur in M_i. A canonical inhabitant of a groupoid law may always be obtained by successive path inductions. Some examples of groupoid laws are given in Figure 1.

Moreover, one can prove that any term M such that $\Gamma \vdash M : c$ is derivable in MLID is extensionally equal to the canonical one. The natural question we answer positively here is: does this uniqueness property of groupoid laws holds in the whole Martin-Löf type theory (MLTT) (with function spaces, universes, sigma types, and inductive families) ? We prove that if $\Gamma \vdash M : c$ is derivable in MLTT then M is equal to the canonical derivation of the groupoid law.

The main idea of the proof is to use successive path inductions to reduce the problem of the uniqueness of inhabitants of a given groupoid law to the uniqueness of the canonical point inhabiting a parametric loop space. Given a base type X and a point $x : X$, the n-th *loop space* and its *canonical point* are inductively defined by:

$$\begin{aligned} \Omega_0(A,a) &:= A & \omega_0(A,a) &:= a \\ \Omega_{n+1}(A,a) &:= \Omega_n(a = a, 1_a) & \omega_{n+1}(A,a) &:= \omega_n(a = a, 1_a) \end{aligned}$$

where $1_a : a = a$ denotes the reflexivity. Thus for any integer n, $\forall X : \mathsf{Type}, x : X . \Omega_n(X,x)$ is a groupoid law inhabited by $\lambda X : \mathsf{Type}, x : X . \omega_n(X,x)$ (note that using one universe, it is possible to internalize the quantification over n; everything that we state here will be true whether or not this is used). We call this groupoid law the n-th *parametric loop space.*

The 0-th parametric loop space, is the polymorphic type $\forall X : \mathsf{Type}. X \to X$ of identity functions, and its canonical inhabitant is $\lambda X : \mathsf{Type}, x : X . x$, ie. the identity function. This term is the only one in MLTT up to function extensionality inhabiting its type. The standard tool to prove this kind of property is by using Reynold's parametricity theory [4] which was introduced to study the behavior of type quantifications within polymorphic λ-calculus (a.k.a. System F). It refers to the concept that well-typed programs cannot inspect types; they must

[1] We believe that Lumsdaine's construction of a contractible globular operad may be described in our framework, but we have not checked it. More generally, a precise study of how models of MLID restricted to contractible contexts compare to definitions of ω-groupoids is out of the scope of the present work.

$$\begin{array}{rcl}
\texttt{id} & : & \forall X : \mathsf{Type}. X \to X \\
\texttt{sym} & : & \forall X : \mathsf{Type}, x : X. x = y \to y = x \\
\texttt{concat} & : & \forall X : \mathsf{Type}, x : X, y : X. x = y \to \forall z : X. y = z \to x = z \\
\texttt{assoc} & : & \forall X : \mathsf{Type}, x : X, y : X, p : x = y, z : X, q : y = z, t : X, r : z = t. \\
 & & \quad \texttt{concat}\, X\, x\, z\, (\texttt{concat}\, X\, x\, y\, p\, z\, q)\, t\, r = \texttt{concat}\, X\, x\, y\, p\, t\, (\texttt{concat}\, X\, y\, z\, q\, t\, r) \\
\texttt{horizontal} & : & \forall X : \mathsf{Type}, x : X, y : X, p : x = y, p' : x = y. p = p' \to \\
 & & \quad \forall z : X, q : x = z, q' : x = z. q = q' \to \texttt{concat}\, X\, x\, y\, p\, z\, q = \texttt{concat}\, X\, x\, y\, p'\, z\, q'
\end{array}$$

Figure 1: Examples of groupoid laws with aliases for canonical inhabitants

behave uniformly with respect to abstract types. Reynolds formalizes this notion by showing that polymorphic programs satisfy the so-called logical relations defined by induction on the structure of types. This tool has been extended by Bernardy et al. [1] to dependent type systems. It provides a uniform translation of terms, types and contexts preserving typing (the so-called *abstraction theorem*). In its unary version (the only needed for this work), logical relations are defined by associating to any well-formed type $A : \mathsf{Type}$ a predicate $[\![A]\!] : A \to \mathsf{Type}$ and to any inhabitant $M : A$ a witness $[\![M]\!] : [\![A]\!]\, M$ that the M satisfies the predicate. This translation may be extended to cope with identity types by taking $[\![a = b]\!] : a = b \to \mathsf{Type}$ to be the predicate defined by $\lambda p : a = b. p_*([\![a]\!]) = [\![b]\!]$ where p_* is the transport along p of the predicate generated by the common type of a and b. Then, it is easy -although quite verbose- to find a translation of introduction and elimination rules of identity types as well as checking that these translations preserve computation rules. This allows to extend the Bernardy's abstraction theorem to identity types. Using this framework, we are able to generalize the uniqueness property of the polymorphic identity type to any parametric loop space. The proof proceed by induction on the dimension of the loop space and uses algebraic properties of transport.

This work shows that parametricity theory may be used to deduce properties about the algebraic structure of identity types. The most interesting question that remains open is whether or not we can extend the translation and the uniqueness property of groupoid laws to deal with Voevodsky's univalence axiom.

References

[1] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free - parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012.

[2] Guillaume Brunerie. Syntactic grothendieck weak ∞-groupoids. Available as `http://uf-ias-2012.wikispaces.com/file/view/SyntacticInfinityGroupoidsRawDefinition.pdf`.

[3] Peter LeFanu Lumsdaine. Weak omega-categories from intensional type theory. In Pierre-Louis Curien, editor, *TLCA*, volume 5608 of *LNCS*, pages 172–187. Springer, 2009.

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

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

[6] Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

Global semantic typing for inductive and coinductive computing

Daniel Leivant

Indiana University
leivant@indiana.edu

Inductive and coinductive types are commonly construed as ontological (Church-style) types, denoting canonical data-sets such as natural numbers, lists, and streams. For various purposes, notably the study of programs in the context of global ("uninterpreted") semantics, it is preferable to think of types as semantical properties (Curry-style).

Intrinsic theories were introduced in the late 1990s to provide a purely logical framework for reasoning about programs and their semantic types [3]. We extend them here to data given by any combination of inductive and coinductive definitions. This approach is of interest because it fits tightly with syntactic, semantic, and proof theoretic fundamentals of formal logic, with potential applications in implicit computational complexity as well as extraction of programs from proofs. We prove a Canonicity Theorem, showing that the global definition of program typing, via the usual (Tarskian) semantics of first-order logic, agrees with their operational semantics in the intended ("canonical") model.

Finally, we show that every intrinsic theory is interpretable in (a conservative extension of) first-order arithmetic. This means that quantification over infinite data objects does not lead, on its own, to proof-theoretic strength beyond that of Peano Arithmetic.

Intrinsic theories are perfectly amenable to formulas-as-types Curry-Howard morphisms, and were used to characterize major computational complexity classes [3, 4, 2, 1]. Their extensions described here have similar potential which has already been applied in [5].

References

[1] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and polytime. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, Perspectives in Computer Science, pages 320–343. Birkhauser-Boston, New York, 1994. www.cs.indiana.edu/∼leivant/papers.

[2] Daniel Leivant. Intrinsic theories and computational complexity. In D. Leivant, editor, *Logic and Computational Complexity*, LNCS, pages 177–194, Berlin, 1995. Springer-Verlag.

[3] Daniel Leivant. Intrinsic reasoning about functional programs I: First order theories. *Annals of Pure and Applied Logic*, 114:117–153, 2002.

[4] Daniel Leivant. Intrinsic reasoning about functional programs II: Unipolar induction and primitive-recursion. *Theor. Comput. Sci.*, 318(1-2):181–196, 2004.

[5] Daniel Leivant and Ramyaa Ramyaa. Implicit complexity for coinductive data: a characterization of corecurrence. In Jean-Yves Marion, editor, *DICE*, volume 75 of *EPTCS*, pages 1–14, 2011.

Proving and Computing with the Harthong-Reeb line using Ω-integers

Nicolas Magaud[1] and Laurent Fuchs[2]

[1] Icube UMR 7357 CNRS - Université de Strasbourg
300 Bld Sébastien Brant BP 10413 67412 Illkirch Cedex, France
magaud@unistra.fr
[2] Laboratoire XLIM-SIC UMR 6172 CNRS - Université de Poitiers
Bat. SP2MI Bld Marie et Pierre Curie
BP 30179 86962 Futuroscope Chasseneuil Cedex, France
Laurent.Fuchs@sic.univ-poitiers.fr

The Harthong-Reeb line offers an alternative model to describe the continuum. It uses only integers to represent real numbers (viewing them at different scales). This is appealing, especially when considering fields of applications such as discrete geometry or geometric computations.

In TYPES 2011, we presented a formal description of the Harthong-Reeb line based on axiomatic non-standard integers (intuitively they correspond to $\mathbb{Z}$ where we introduce a - new - infinitely large number ω). We formally proved using Coq [5] that this description of the Harthong-Reeb line does satisfy all the axioms for constructive real numbers proposed by D. Bridges [1].

However, having an axiomatic description of the underlying non-standard integers prevents us from computing in this formalism. Thus it has been investigated how to implement these non-standard integers using Laugwitz-Schmieden integers [2].

Laugwitz-Schmieden integers (also known as Ω-integers) are sequences of elements of $\mathbb{Z}$. This representation of non-standard integers allows to build a constructive description of the Harthong-Reeb line. Thus, we can implement the Ω-arithmetization scheme (adapted from Euler's one) proposed in [3]. Using extraction in Ocaml and the graphical interface of Ocaml, we obtain discrete representations of *continuous* functions at different scales as shown in Fig. 1.

One of our goals is to establish the correction of this Ω-arithmetization scheme for continuous functions as Fleuriot does in [4] using hyperreals. To do so, we must first establish that the Harthong-Reeb line based on Laugwitz-Schmieden integers verifies Bridges' axioms.

As showed by Chollet et al. in [2], most properties of the constructive real line hold for this model based on Laugwitz-Schmieden (except three of them: two properties about the order and the least upper bound principle). These restrictions come from the fact that Ω-integers are a very rich structure where many Ω-integers have no interpretation as naive (or standard) integers. Chollet et al. thus propose an alternative axiom system which is very close to Bridges' one, but defines an alternative continuum.

We formally proved using Coq that the description of this continuum based on Ω-integers actually verifies all of these (new) axioms. Work in progress consists in finding a way to characterize a subset of the Harthong-Reeb line based on Ω-integers which corresponds to constructive real numbers (such as those available in CoRN). We then expect to prove using Coq that this subset is actually isomorphic to the constructive real numbers of CoRN.

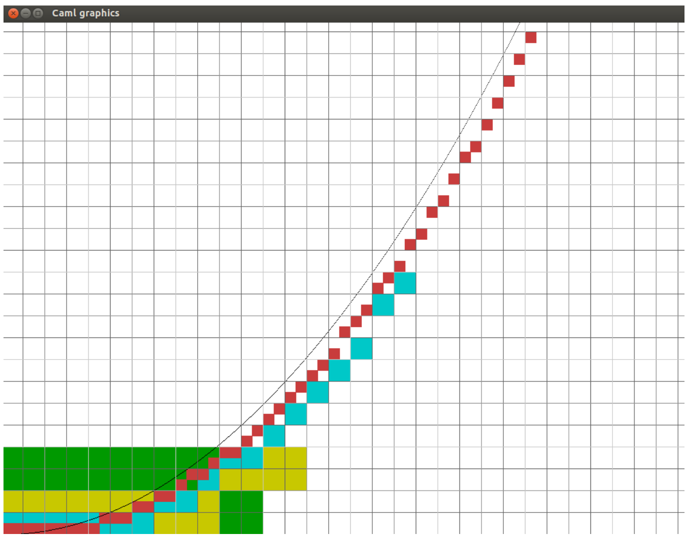


Figure 1: The arithmetization of the function $t \mapsto \frac{t^2}{6}$. Graphs of the function $\widetilde{t} \mapsto \widetilde{x}(\widetilde{t})$ are drawn at different ranks of the involved infinite integer sequences.

References

[1] D. S. Bridges. Constructive mathematics: A foundation for computable analysis. *Theor. Comput. Sci.*, 219(1-2):95–109, 1999.

[2] A. Chollet, G. Wallet, L. Fuchs, E. Andres, and G. Largeteau-Skapin. Foundational Aspects of Multiscale Digitization. *Theor. Comput. Sci.*, 466:2–19, 2012.

[3] A. Chollet, G. Wallet, L. Fuchs, G. Largeteau-Skapin, and E. Andres. Insight in discrete geometry and computational content of a discrete model of the continuum. *Pattern recognition*, 42:2220–2228, 2009.

[4] J. Fleuriot. Exploring the foundations of discrete analytical geometry in Isabelle/HOL. In P. Schreck, J. Richter-Gebert, and J. Narboux, editors, *Proceedings of Automated Deduction in Geometry 2010*, volume 6877 of *LNAI*. Springer, 2011.

[5] N. Magaud, A. Chollet, and L. Fuchs. Formalizing a Discrete Model of the Continuum in Coq from a Discrete Geometry Perspective. *Submitted to an international journal.*, 2013. see `https://dpt-info.u-strasbg.fr/~magaud/Harthong-Reeb/`.

A Kleene realizability semantics for the Minimalist Foundation

Maria Emilia Maietti and Samuele Maschio

Dipartimento di Matematica, University of Padova, Italy

Abstract

The Minimalist Foundation was ideated by M. E. Maietti and G. Sambin in [MS05] and then completed in [Mai09] by M. E. Maietti. It is intended to constitute a common core among the most relevant constructive and classical foundations. One of its novelties is that it consists of two levels: an intensional level (mTT) which should make evident the constructive contents of mathematical proofs in terms of programs, and an extensional level (emTT) formulated in a language close as much as possible to that of ordinary mathematics. Both the intensional level and the extensional level of the Minimalist Foundation consist of type systems based on versions of Martin-Lof's type theory with the addition of a primitive notion of propositions: the intensional one is based on [NPS90] and the extensional one on [Mar84].

In this talk we show how to build a predicative realizability model of the Minimalist Foundation, and in particular of its extensional level emTT, validating the Extended Church thesis (ECT).

To reach this goal it is enough to build a realizability model for the intensional level mTT validating ECT. Indeed a realizability interpretation for the extensional level emTT can be then obtained from an interpretation of mTT by composing this with the interpretation of emTT in a suitable setoid model of mTT as in [Mai09] and analyzed in [MR13].

We build the realizability model for mTT+ECT in the theory $\widehat{ID_1}$([Fef82]). This theory is formulated in the language of second-order arithmetics and it consists of PA (Peano Arithmetic) plus the existence of some (not necessary the least) fixed point for positive parameter-free arithmetical operators. Our realizability model is obtained by suitably modifying the realizability semantics in $\widehat{ID_1}$ described in [Bee85] for the extensional version of first-order Martin-Löf's type theory with one universe, which is based on Kleene realizability semantics of intuitionistic arithmetics.

In essence we interpret mTT-sets as Beeson interpreted Martin-Löf's sets following Kleene realizability in $\widehat{ID_1}$, propositions are interpreted as proof-irrelevant quotients of their Kleene realizability interpretation and the universe of mTT-small propositions as a suitable quotient of some fix-point including all the codes of small propositions.

It is worth to recall that our modifications to Beeson's model are essential, because Beeson's model for the extensional version of Martin-Löf's type theory can not validate ECT due to the inconsistency of the full axiom of choice and function extensionality with it. If we drop function extensionality and we take the intensional version of Martin-Löf's type theory then this version might be consistent with ECT, and from it we can easily derive the consistency of mTT+ECT+ full axiom of choice, but this is still an open problem.

References

[Bee85] M. Beeson. *Foundations of Constructive Mathematics.* Springer-Verlag, Berlin, 1985.

[Fef82] S. Feferman. Iterated inductive fixed-point theories: application to Hancock's conjecture. In *Patras Logic Symposion*, pages 171–196. North Holland, 1982.

[Mai09] M. E. Maietti. A minimalist two-level foundation for constructive mathematics. *Annals of Pure and Applied Logic*, 160(3):319–354, 2009.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory. Notes by G. Sambin of a series of lectures given in Padua, June 1980.* Bibliopolis, Naples, 1984.

[MR13] M. E. Maietti and G. Rosolini. Quotient completion for the foundation of constructive mathematics. *Logica Universalis*, 7(3):371–402, 2013.

[MS05] M. E. Maietti and G. Sambin. Toward a minimalist foundation for constructive mathematics. In L. Crosilla and P. Schuster, editor, *From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics*, number 48 in Oxford Logic Guides, pages 91–114. Oxford University Press, 2005.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin Löf's Type Theory.* Clarendon Press, Oxford, 1990.

Polymorphic variants in dependent type theory

Claudio Sacerdoti Coen^{1*} and Dominic 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. The problem consists in the extension of a data type of expressions to include new constructors. All operations defined on the data type have to be updated to cover also the new constructors. The difficulty is that the extension should be done modularly: it should be possible to add new constructors or to merge together two sets of them without modifying the already developed code. Object oriented languages provide a solution to the problem: the type of expressions becomes an interface that is implemented by each constructor, represented as a class. The tradeoff is the impossibility to close the universe of expressions in order to perform 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 in "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 $E_2\ \alpha$ is it 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 disallows multiple inheritance. Finally, the solution is not efficient since every merge operation adds an indirection that is paid for 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 [**?**, **?**] . 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. Replaying 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

*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.

example, a function could be typed as $[K_1 : T_1 \mid K_2 : T_2] \cup \rho \rightarrow T$ to mean that the input can be any type obtained by 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 show 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 name bounded polymorphic variants 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 modifications. Therefore, the restriction to the bounded case is not problematic because 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 discussed in the literature must be derivable. In particular, each bounded polymorphic variant should come with its own elimination principle that allows 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 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 to turn the two lists into the sigma-type of inhabitants of the universe that are built only from constructors that respect the constraints; code extraction turns the sigma-type into the universe type, ensuring efficiency; implicit coercions are used to hide the sigma 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.

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

Some Varieties of Constructive Finitenes

Erik Parmann

Introduction

In this note we consider two related notions of *constructive finiteness*. First we look at how we can define that the set of **True** positions in a stream over bool is finite. This extends the work done by Bezem, Nakata and Uustalu [1] by identifying a new formalisation which lies between two known ones, and identifying that reductions ”upwards” in the hierarchy is equivalent to *Markov's Principle* and the *Weak Limited Principle of Omniscience*, respectively.

In the second part we look at *streamless sets*, recently investigated by Coquand and Spiwack [2]. A set is *streamless* if, intuitively, every stream over that set must contain duplicates. It is an open question whether the product of two streamless sets is a streamless set. Here we show that this indeed holds if we assume Markov's Principle and decidable equality.

Definitions

We start by providing the basic definitions. A stream over A is a function of type $\mathbb{N} \to A$. For a stream $f : \mathbb{N} \to \text{bool}$, $\texttt{NrOfTrue}_f : \mathbb{N} \to \mathbb{N}$ is the function which on input k returns the number of natural numbers i such that $i \leq k$ and $f(i) = \textbf{True}$, that is, $\texttt{NrOfTrue}_f\ k = |\{i \mid i \leq k \wedge f(i) = \textbf{True}\}|$.

Now we provide three different properties that can hold of Boolean streams. A function $f : \mathbb{N} \to \text{bool}$ is *eventually always false*, written $\text{eaf}(f)$ when

$$\exists n : \mathbb{N}, \forall m : \mathbb{N}, m \geq n \to f(m) = \textbf{False}\,.$$

Bezem et al. [1] refers to this as both Equation (1) and ”$\mathcal{F}(\mathcal{G}\text{ blue})s$”.

Next, we say that a stream $f : \mathbb{N} \to \text{bool}$ is *bounded*, written $\text{bounded}(f)$ when there is a bound to the number of **True** positions:

$$\exists n : \mathbb{N}, \forall k : \mathbb{N}, \texttt{NrOfTrue}_f\ k \leq n.$$

Bezem et al. [1] refers to this as both Equation (2) and ”$\exists n.\, \text{le}_n\, s$”.

The new notion is a natural strengthening of bounded. We say that a stream $f : \mathbb{N} \to \text{bool}$ is *strictly bounded*, written $\text{sb}(f)$ when there exists a strict bound on the number of **True** positions:

$$\exists n : \mathbb{N}, (\forall k : \mathbb{N}, \texttt{NrOfTrue}_f\ k \leq n \wedge \neg\forall k : \mathbb{N}, \texttt{NrOfTrue}_f\ k \leq n - 1).$$

Finally we have three axioms which are all constructively consistent, but not provable. Markov's Principle (MP) states that for any decidable predicate over natural numbers (i.e., streams over bool): $\neg\neg(\exists n : \mathbb{N}, Pn) \to \exists n : \mathbb{N}, Pn$. Intuitively MP is realized by an unbounded search, which we know (from outside the system) will terminate because of the antecedent.

Weak Limited Principle of Omniscience (WLPO) states that for any decidable predicate P, we have that $(\forall n : \mathbb{N}, P(n)) \vee (\neg\forall n : \mathbb{N}, P(n))$, while the stronger Limited Principle of Omniscience (LPO) states that $(\forall n : \mathbb{N}, P(n)) \vee (\exists n : \mathbb{N}, \neg P(n))$. It is rather easy to see that $(MP \wedge WLPO) \Rightarrow LPO$, and in fact we have the stronger $(MP \wedge WLPO) \iff LPO$.

Relations between the formulae

Bezem et al. [1] showed that for any stream f we have $\mathrm{eaf}(f) \Rightarrow \mathrm{bounded}(f)$. It is obvious that $\mathrm{sb}(f) \Rightarrow \mathrm{bounded}(f)$, and $\mathrm{eaf}(f) \Rightarrow \mathrm{sb}(f)$ is also clear, as the bound on the index in eaf gives us a bound where all the **True** values must reside, letting us find the exact amount of **True**'s. They furthermore show that $(\forall f, \mathrm{bounded}(f) \Rightarrow \mathrm{eaf}(f)) \iff LPO$.

We find that the new notion of strictly bounded falls neatly between the two previous notions, completing the picture:

Lemma 1. $(\forall f, \mathrm{sb}(f) \Rightarrow \mathrm{eaf}(f)) \iff MP$.

Lemma 2. $(\forall f, \mathrm{bounded}(f) \Rightarrow \mathrm{sb}(f)) \iff WLPO$.

As neither MP, WLPO or LPO hold constructively, we get a strict hierarchy where eaf $\Rightarrow$ sb and sb $\Rightarrow$ bounded holds constructively, but none of the other directions hold without further assumptions.

Streamless

In this section we will look at streamless sets. A set B is streamless if for all streams $g : \mathbb{N} \to B$ we have indices $i < j$ and $g(i) = g(j)$. See Coquand and Spiwack [2] for further elaborations. It is an open problem whether $A \times B$ is streamless whenever A and B are. Here we look at two special cases in which it holds.

First we observe several properties of streamless sets. If we have a stream g over streamless B we can make a new stream over $B \times \mathbb{N}$, such that for every $\langle b, i\rangle$ in the new stream, b occurs at least twice in g. We get this letting it begin with the pair $\langle g(j), j\rangle$ with j as above, and then continuing likewise on the stream we get by starting g at index j. We can iterate this process, and for every $n : \mathbb{N}$ we get a stream such that every element in the new stream gives a $b : B$ and an index such that b occurs at least n times in g before the index. Given a stream $g : \mathbb{N} \to B$ we denote by $g^x : \mathbb{N} \to B \times \mathbb{N}$ the stream which gives pairs $\langle b, i\rangle$ such that there is at least x occurrences of b before i in g.

Finally, we say that a set A is bounded by $n : \mathbb{N}$ if every list over A with more than n elements must contain duplicates.

Lemma 3. *If A is bounded by n and B is streamless then $A \times B$ is streamless.*

Proof sketch. Assuming $g : \mathbb{N} \to A \times B$ we can look at $g_2 : \mathbb{N} \to B$, its second projection. By looking at $(g_2)^{n+1}(0)$ we get a pair $\langle b, i\rangle$ such that b occurs at least $n + 1$ times before i in g_2. As this is a bounded range we can extract the list $[j_1, \ldots, j_i]$ of indices where b occurs in g_2. Note that $[g_1(j_1), \ldots, g_1(j_i)]$ is $n+1$ elements of A, so there must be at least two indices $j_k < j_l$ such that $g_1(j_k) = g_1(j_l)$. As $g_2(j_k) = b = g_2(j_l)$ we get $g(j_k) = g(j_l)$. □

Lemma 4. *Assuming Markov's Principle and decidable equality on A we have that $A \times B$ is streamless whenever A and B are.*

Proof sketch. Assume a stream $g : \mathbb{N} \to A \times B$. We then define the predicate $P(n) :=$"for $\langle b, i\rangle = (g_2)^n(0)$ we have duplicates in the list generated from taking the A-elements from the n pairs before i with b as their second element." Notice that $P(n)$ is decidable as long as B is streamless and A has decidable equality.

We want to prove $\neg\neg\exists n, P(n)$ when A is streamless. From $\neg\exists n, P(n)$ we get that for every n we have a list of n elements of A without duplicates. This allows us to construct a stream over A without duplicates, contradicting that A is streamless.

Using Markov's Principle we get an n such that $P(n)$, and by essentially the same argument as for Lemma 3 we get the two indices $i < j$ with $g(i) = g(j)$. $\square$

References

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

[2] Thierry Coquand and Arnaud Spiwack. Constructively finite? In *Contribuciones científicas en honor de Mirian Andrés Gómez*, pages 217–230. Universidad de La Rioja, 2010.

A Calculus of Primitive Recursive Constructions

Hugo Herbelin and Ludovic Patey

Université Paris Diderot, Paris, France
Hugo.Herbelin@inria.fr
Ludovic.Patey@computablity.fr

Primitive Recursive Arithmetic (PRA) is a predicative formal subsystem of Peano Arithmetic, which aims to capture finitist reasoning. PRA is expressive enough to manipulate basic structures using codings to natural numbers. The tradeoff between its expressive power and the consensus on finitist reasoning makes it a good candidate as a meta-system in proof theory [3].

However the necessity to go through coding for manipulating any structure different from natural numbers makes proofs more tedious. Moreover, a wide class of formal structures can be easily proven to be encodable within PRA, so the restriction to natural numbers as the only primitive structure of the language is not justified. It becomes clear that proof theory would benefit from the introduction of a typed language enabling mathematicians to express in a natural way their objects manipulated. Such a language should be proven to be able to express only structures encodable in natural numbers within Primitive Recursive Arithmetic.

With the advent of computers, formal proof assistants have been developed [2, 5], answering to the request of more reliability in the proofs, enabling more collaboration in the research process and therefore to tackle bigger projects. Designed to capture the broadest audience, they provide very expressive formal languages to be able to mechanise in a natural way a wide range of proofs. This policy finds its limits in proof theory where mathematicians want to ensure that their proofs are formalisable in a weak formal system. A proof theorist may want to parametrise the expressiveness of his proof assistant, leading to a syntactical restriction of the formal language.

We design a Calculus of Primitive Recursive Constructions (CPRC) as a subsystem of the Calculus of Inductive Constructions without dependent product. In practice, in our approach of CPRC, inductive types are presented algebraically from the type constructors 0, 1, $+$, Σ, $=$ and μ, while recursion is decomposed into pattern-matching and guarded fixpoint as done in [4]. We provide two translations α and β between a logic-free presentation of PRA [1] and CPRC. We also provide proofs of their correctness in a sense defined below. Hence the CPRC has the same expressiveness as PRA, while being able to express natural structures in a simpler way.

Let Nat be the inductive type $\mu X.1 + X$ and $=_{Nat}$ be the equality over Nat.

$$t_1 = u_1, \ldots, t_n = u_n \vdash_{PRA} t = u$$

is a valid PRA judgement iff there exists a proof term p such that

$$x_1 : [\![t_1]\!]_\alpha =_{Nat} [\![u_1]\!]_\alpha, \ldots, x_n : [\![t_n]\!]_\alpha =_{Nat} [\![u_n]\!]_\alpha \vdash_{CPRC} p : [\![t]\!]_\alpha =_{Nat} [\![u]\!]_\alpha$$

is a valid CPRC judgement. Conversely, terms of CPRC are coded as terms in PRA whereas types are interpreted as characteristic functions of the coding of their inhabitants. If

$$x_1 : A_1 \ldots, x_n : A_n \vdash_{CPRC} p : A$$

is a valid CPRC judgement, then

$$[\![A_1]\!]_\beta\ x_1 = 0, \dots, [\![A_n]\!]_\beta\ x_n = 0 \vdash_{PRA} [\![A]\!]_\beta\ [\![p]\!]_\beta = 0$$

is a valid PRA judgement. The conditions under which the converse holds are still under study.

A CPRC judgement is a refinement of a PRA judgement as it contains the proof terms, hence is more informative about the derivation tree. Note that in CPRC as in CIC, there is no distinction between formulae and types. In particular equality is a type.

This calculus is a first step to an implementation in Coq, opening the door to proofs within Primitive Recursive Arithmetic using formal proof assistants.

References

[1] Haskell B Curry. A formalization of recursive arithmetic. *American Journal of Mathematics*, pages 263–282, 1941.

[2] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Benjamin Werner, Christine Paulin-Mohring, et al. The coq proof assistant user's guide: Version 5.6. 1991.

[3] Gerhard Gentzen. Die widerspruchsfreiheit der reinen zahlentheorie. *Mathematische Annalen*, 112(1):493–565, 1936.

[4] Hugo Herbelin and Arnaud Spiwack. The rooster and the syntactic bracket. *arXiv preprint arXiv:1309.5767*, 2013.

[5] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

Liquid Types Revisited

Mário Pereira, Sandra Alves, and Mário Florido

University of Porto, Department of Computer Science & LIACC,
R. do Campo Alegre 823, 4150-180, Porto, Portugal

Abstract

We present a new type system combining refinement types ideas and the expressiveness of intersection type discipline. The use of such features makes it possible to derive very precise types, using the types language itself as a detailed description for programs' functional behaviour. We have been able to prove several interesting properties for our system (including subject reduction) and started the development of an inference algorithm, which was proved sound.

1 Motivation

Refinement types [2, 4] state complex program invariants, by augmenting type systems with logical predicates. A refinement type of the form $\{\nu : B \mid \phi\}$ stands for the set of values from basic type B restricted to the filtering predicate (refinement) ϕ. A subtyping relation exists for refinement types, which will generate implication conditions (much like a VCGen in the context of program verification):

$$\frac{\Gamma; \nu : B \vdash \phi \Rightarrow \psi}{\Gamma \vdash \{\nu : B \mid \phi\} <: \{\nu : B \mid \psi\}}$$

One idea behind the use of such type systems is to perform type-checking using SMTs (Satisfability Modulo Theories), discharging conditions as the above $\phi \Rightarrow \psi$. However, the use of arbitrary boolean terms as refinement expressions leads to undecidable type systems, both for type checking and inference.

Liquid Types [5, 6] (*Logically Qualified Data Types*) present a system capable of automatically infer refinement types, by means of two main restrictions to a system: every refinement predicate is a conjunction of expressions exclusively taken from a global, user-supplied set (denoted $\mathbb{Q}$) of logical qualifiers (simple predicates over program variables, the value variable ν and the variable placeholder $\star$); and a conservative (hence decidable) notion of subtyping.

The Liquid Types system is defined as an extension to the Damas-Milner type system, with the term language extended with an `if-then-else` constructor and constants. A key idea behind this system is that the refinement type of every term is a refinement of the corresponding ML type.

Despite the interest of Liquid Types, some situations arise where the inference procedure infers poorly accurate types. For example, considering $\mathbb{Q} = \{\nu \geq 0, \nu \leq 0\}$ and the term neg $\equiv \lambda x. -x$, Liquid Types system infers neg $:: x : \{\nu : int \,|\, 0 \leq \nu \wedge 0 \geq \nu\} \rightarrow \{\nu : int \,|\, 0 \leq \nu \wedge 0 \geq \nu\}$ (the syntax $x : \tau \rightarrow \sigma$ is here preferred over the usual $\Pi(x : \tau).\sigma$ for functional dependent types). This type cannot, at all, be taken as a precise description of the neg function's behaviour.

2 Intersection-refinement types

We propose a refinement type system with the addition of intersection types [1]. Our intersections are at the refinement expressions level only, i.e. for the type $\sigma \cap \tau$ both σ and τ are of the same form, solely differing in the refinement predicates. We introduce a new rule to form intersections on types (our typing relation is denoted by $\vdash^{\cap}$):

$$\textsc{Intersect}\quad \frac{\Gamma \vdash^{\cap} M : \sigma \qquad \Gamma \vdash^{\cap} M : \tau}{\Gamma \vdash^{\cap} M : \sigma \cap \tau}$$

As an example, the neg function could be typed within our system as

$$(x : \{\nu : int \mid \nu \geq 0\} \rightarrow \{\nu : int \mid \nu \leq 0\}) \cap (x : \{\nu : int \mid \nu \leq 0\} \rightarrow \{\nu : int \mid \nu \geq 0\})$$

Our use of intersections for refinement types draws some inspiration from [3].

We use a standard call-by-value small step operational semantics to define the evaluation relation, denoted $\rightsquigarrow$. Based on this relation, we were able to prove the following result:

Theorem 1 (Subject reduction). *If* $\Gamma \vdash^{\cap} M : \sigma$ *and* $M \rightsquigarrow N$ *then* $\Gamma \vdash^{\cap} N : \sigma$.

Based on the Liquid Types restrictions, we conceived an algorithm to infer appropriate refined types with intersections. We use the set $\mathbb{Q}$ to restrict the possible refinements of types and to guarantee that the algorithm terminates. We define the inference algorithm in terms of a program M, a typing context Γ and $\mathbb{Q}$ as $\texttt{Infer}(\Gamma, M, \mathbb{Q}) = \sigma$, where σ is an intersection-refined type. The following result holds:

Theorem 2 (Soundness). *If* $\texttt{Infer}(\Gamma, M, \mathbb{Q}) = \sigma$ *then* $\Gamma \vdash^{\cap} M : \sigma$.

With the use of intersections we are able to derive more precise types than in a classical refinement type system. These types can thus be taken as detailed descriptions of programs' behaviour.

We are currently investigating completeness properties of our inference algorithm. We take particular interest in realizing if our algorithm infers *most-general types* by means of subtyping. In other words, we would like to prove that if $\texttt{Infer}(\Gamma, M, \mathbb{Q}) = \sigma$ and $\Gamma \vdash^{\cap} M : \sigma'$, for some σ', then $\sigma <: \sigma'$.

References

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

[2] Ewen Denney. Refinement types for specification. In *Programming Concepts and Methods PROCOMET'98*, pages 148–166. Springer, 1998.

[3] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.

[4] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, February 2010.

[5] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

[6] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 209–228, Berlin, Heidelberg, 2013. Springer-Verlag.

Toward a New Formulation of Extensional Type Theory

Andrew Polonsky

VU Amsterdam

We report progress on the design of a type system with extensional equality that would admit decidable type checking by avoiding the propositional reflection rule.

Our approach is similar to Observational Type Theory of Altenkirch et al, in that extensional equality is generated from a logical relation defined by induction on type structure. Equality on Π-types is defined to be pointwise equality of functions.

Our contribution is to reflect the logical relation by a dedicated type constructor (which inhabitants represent *type equalities*) in such a way that extensionality witnesses exist for every term *in the same type universe.* This allows us to define a higher-dimensional substitution operation:

$$\frac{\Gamma, x : A \vdash t : T \qquad \Gamma \vdash a : A}{\Gamma \vdash t[a/x] : T[a/x]} \qquad\qquad \frac{\Gamma, x : A \vdash t : T \qquad \Gamma \vdash a^* : a \simeq_A a'}{\Gamma \vdash t[a^*//x] : t[a/x] \sim_{T[a^*//x]} t[a'/x]}$$

The fact that T and $t[a] \sim_{T[a^*]} t[a']$ belong to the same type hierarchy allows the operation to be iterated to arbitrary dimensions.

Every type has the structure of a globular set with reflexivities, which furthermore satisfies a certain kind of Kan filling condition. The condition is realized by new operators which formally witness the homotopy lifting property.

For simplicity, we present our system as an extension of $\lambda{*}$, with the understanding that the stratified version is conjectured to be strongly normalizing.

$$\begin{array}{rcl} A, t, e, \gamma & ::= & * \mid x \mid \Pi x{:}A.B \mid \Sigma x{:}A.B \mid A \simeq B \mid a \sim_e b \mid a \simeq_A a' \\ & & \mid \lambda x{:}A.t \mid st \mid (s,t) \mid \pi_1 t \mid \pi_2 t \\ & & \mid *^* \mid \Pi^*[x,x',x^*]{:}A^*.B^* \mid \Sigma^*[x,x',x^*]{:}A^*.B^* \mid \simeq^* A^* B^* \\ & & \mid \mathsf{r}(t) \mid e(t) \mid \bar{e}(t) \mid t_e \mid t^e \mid \gamma \looparrowdownleft_{e^*} \gamma' \end{array}$$

This system has three kinds of equality relations.

The constructor $A \simeq B$ represents the type of equalities between types:

$$\frac{A : * \qquad B : *}{A \simeq B : *}$$

Any term $e : A \simeq B$ of this type induces a binary relation between the corresponding types:

$$\frac{e : A \simeq B \qquad a : A \qquad b : B}{a \sim_e b : *}$$

The term former $\sim_e : A \to B \to *$ can thus be seen as the eliminator for the type $A \simeq B$. The constructors for this type are the symbols $*^*, \Pi^*, \Sigma^*, \simeq^*$, which witness the fact that equality is a congruence with respect to all type constructors (including $\simeq$ itself).

The type $A \simeq B$ enjoys the following computation rules:

$$\begin{array}{rcl} A \sim_{*^*} B & \longrightarrow & A \simeq B \\ f \sim_{\Pi^*[x,x',x^*]:A^*B^*} f' & \longrightarrow & \Pi a{:}A \Pi a'{:}A' \Pi a^* : a \sim_{A^*} a'.\ fx \sim_{B^*[a/x,a'/x',a^*/x^*]} f'x' \\ p \sim_{\Sigma^*[x,x',x^*]:A^*B^*} p' & \longrightarrow & \Sigma a^* : \pi_1 p \sim_{A^*} \pi_1 p'.\ \pi_2 p \sim_{B^*[\pi_1 p/x, \pi_1 p'/x', a^*/x^*]} \pi_2 p' \\ e \sim_{\simeq^* A^* B^*} e' & \longrightarrow & \Pi a{:}A \Pi a'{:}A' \Pi a^* : a \sim_{A^*} a' \\ & & \Pi b{:}B \Pi b'{:}B' \, \Pi b^* : b \sim_{B^*} b'.\ (a \sim_e b) \simeq (a' \sim_{e'} b') \end{array}$$

The type $a \simeq_A a'$ is the extensional equality type on A:

$$\frac{A : * \qquad a : A \qquad a' : A}{a \simeq_A a' : *} \qquad\qquad \frac{a : A}{\mathsf{r}(a) : a \simeq_A a}$$

The following identities are valid:

$$\begin{array}{rcll} t[a^*//x] & = & \mathsf{r}(t) & \text{if } x \notin \mathsf{FV}(t) \\ a \simeq_A a' & = & a \sim_{\mathsf{r}(A)} a' & \\ A \simeq B & = & A \simeq_* B & \end{array}$$

In particular, $(A \simeq B) = (A \simeq_* B) = (A \sim_{\mathsf{r}(*)} B) = (A \sim_{*[a^*//x]} B)$, justifying the typing of the higher-dimensional substitution.

For example, one can define in this system the "mapOnPaths" operation

$$\frac{\Gamma \vdash t : \Pi x{:}A.B \qquad \Gamma \vdash \alpha : a_1 \simeq_A a_2}{\Gamma \vdash t.\alpha : ta_1 \sim_{B[\alpha//x]} ta_2}$$

by taking $t.\alpha \coloneqq \mathsf{r}(t)a_1a_2\alpha$. It computes as

$$(\lambda x{:}A.b).\alpha \quad \longrightarrow \quad b[\alpha//x]$$

So far, the equality type gives each type the structure of a globular set with reflexivities. For higher-dimensional compositions, this structure must also satisfy a certain filling condition. We obtain this by adding operations which allow one to transfer terms from one side of type equality to another.

$$\begin{array}{cc} \dfrac{e : A \simeq B \qquad a : A}{e(a) : B} & \qquad \dfrac{e : A \simeq B \qquad b : B}{\bar{e}(b) : A} \\ a_e : a \sim_e e(a) & \qquad b^e : \bar{e}(b) \sim_e b \end{array}$$

These are operations with reduction rules. As they can be used in any context, they have the effect of witnessing the homotopy lifting property:

$$\frac{\Gamma, x : A \vdash B(x) : * \qquad \Gamma \vdash a^* : a \simeq_A a' \qquad \Gamma \vdash b : B(a)}{\Gamma \vdash b_{B(a^*)} : b \sim_{B(a^*)} B(a^*)(b)}$$

where $B(a^*) = B[a^*//x] : B(a) \simeq B(a')$.

Composition and symmetry are defined as follows. For $\alpha : a \simeq_A a'$, we have

$$\begin{array}{ll} \mathring{\alpha}(x) : (x \simeq_A a) \simeq (x \simeq_A a') & \qquad \overline{\alpha} : a' \simeq_A a \\ \mathring{\alpha}(x) \coloneqq (x \simeq_A y)[\alpha//y] & \qquad \overline{\alpha} \coloneqq \overline{\mathring{\alpha}(a')}(\mathsf{r}(a')) \end{array}$$

In particular, for $a^* : a_0 \simeq_A a$, we have $\mathring{\alpha}(a_0)(a^*) : a_0 \simeq_A a'$.

Higher-dimensional fillers can be constructed following this pattern.

The $\looparrowdown_{e^*}$ operation is an "exchange law" used when higher cells are substituted over higher cells, as, for example, in the clause for path substitution over reflexivity:

$$\mathsf{r}(B)[a^*//x] \quad = \quad \looparrowdown_{\mathsf{r}(B[a^*//x])}$$

Dialectica: From Gödel to Curry-Howard

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`

Originally introduced by Gödel in the eponymous *Dialectica* journal in 1958 [6], the Dialectica transformation was a tentative workaround to the then-perceived cataclysm of the incompleteness theorems. As classical logic could not be considered a firm ground for the foundations of mathematics anymore, one had to rely upon constructive arguments.

Similarly to its predecessor, the double-negation translation, Dialectica aimed at providing classical logic with computational roots, through a transformation of **HA** into system **T** [1]. Unlike the double-negation translation, Dialectica is more fine-grained. Indeed, while retaining the disjunction and existence properties of intuitionistic logic, Dialectica realizes two semi-classical principles, namely Markov's principle (MP) and independence of premises (IP), which, given any decidable proposition P on natural numbers, are usually stated as:

$$\text{MP}\ \frac{\neg(\forall n^{\mathbb{N}}.\, \neg P\, n)}{\exists n^{\mathbb{N}}.\, P\, n} \qquad \frac{(\forall m^{\mathbb{N}}.\, P\, m) \to \exists n^{\mathbb{N}}.\, R\, n}{\exists n^{\mathbb{N}}.\, (\forall m^{\mathbb{N}}.\, P\, m) \to R\, n}\ \text{IP}$$

While representing a major breakthrough at the time of its publication, the Dialectica transformation looks rather *bizarre* by modern standards of proof theory. First, as already observed by then [3], the translation of the contraction rule required the atomic propositions to be decidable. Second, and more worrisome in the Curry-Howard paradigmatic view, the translation does not preserve β-equivalence.

In her PhD thesis [2], De Paiva proposed a categorical presentation of the Dialectica translation that somehow solved both issues at once. This presentation made the crucial observation that the original Dialectica could be understood as a translation acting over linear logic rather than intuitionistic logic, using Girard's historical call-by-name decomposition of the arrow [4]. This categorical presentation led to more intricate constructions [9, 8], allowing both to factorize models of linear logic from the literature through this Dialectica-like transformation, and to easily design new ones by following the same pattern.

Conversely, and strangely enough, to the best of our knowledge, the Dialectica translation by itself did not benefit from this categorical apparatus. In particular, a clear understanding of the computational effects at work in the translation remained to be found. What does the program corresponding to the translation of a proof actually do?

We answer this legitimate question by providing our own syntactical, untyped presentation of Dialectica in a slightly extended λ-calculus, based on De Paivas's work. It happens that the very computational content of this translation can be easily explained thanks to the usual Krivine abstract machine (KAM) with closures, in an approach quite similar to the one of classical realizability [10, 12]. Essentially, Dialectica allows to capture the current stack of the machine when accessing a variable in the environment.

This feature can be seen as a weak form of delimited control, embodied by the operator

$$\mathcal{M} : (A \Rightarrow B) \Rightarrow A \Rightarrow {\sim}B \Rightarrow \mathfrak{M}({\sim}A)$$

where ${\sim}X$ denotes the type of stacks of X and $\mathfrak{M}X$ the finite multiset over X. Here, stacks are given a first-class citizenship and made inspectable, which is fairly stronger than the usual arrow type of continuations. Given any function $f : A \Rightarrow B$, any argument $t : A$ and any return stack $\pi : {\sim}B$, $\mathcal{M}\, f\, t\, \pi$ computes the multiset of stacks $\{\rho_1, \ldots, \rho_n\}$ where each ρ_i is the current stack of the machine for each corresponding use of t by f, delimited by π.

There is an intriguing mismatch, though. Indeed, the KAM produces the stacks in a definite order, because of the sequentiality of the reduction, but the Dialectica translation does not, because it constructs a multiset instead of a list. Yet, there is no obvious way to tweak the Dialectica transformation to recover the sequentiality in the list of produced stacks. This defect actually seems deeply rooted in the linear decomposition itself.

Our syntactical presentation has the advantage to be compatible with the usual constructions around λ-calculus. For instance, we can easily apply it to more complicated settings, like dependent types. We obtain a Dialectica translation for $\mathbf{CC}_\omega$ [11] almost trivially. The translation is also applicable to the dependent elimination of inductives, hinting towards a translation for the full-fledged **CIC** system.

In a more general way, the dependently-typed Dialectica gives interesting hindsights into what could be (or not) linear dependent types, and provides more generally enlightening intuitions about effects and continuations in a dependent type theory. Finally, we believe that we could design similar transformations inspired by its computational content able to provide well-behaved versions of delimited control.

References

[1] Jeremy Avigad and Solomon Feferman. Gödel's functional (”Dialectica”) interpretation, 1998.

[2] Valeria de Paiva. A dialectica-like model of linear logic. In *Category Theory and Computer Science*, pages 341–356, 1989.

[3] Justus Diller. Eine Variante zur Dialectica-Interpretation der Heyting-Arithmetik endlicher Typen. *Archiv für mathematische Logik und Grundlagenforschung*, 16(1-2):49–66, 1974.

[4] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[5] Jean-Yves Girard. *The Blind Spot: Lectures on Logic.* European Mathematical Society, 2011.

[6] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[7] Hugo Herbelin. An intuitionistic logic that proves Markov's principle. *LICS*, pages 50–56, 2010.

[8] J. M. E. Hyland. Proof theory in the abstract. *Ann. Pure Appl. Logic*, 114(1-3):43–78, 2002.

[9] Martin Hyland and Andrea Schalk. Glueing and orthogonality for models of linear logic. *Theor. Comput. Sci.*, 294(1/2):183–231, 2003.

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

[11] Zhaohui Luo. An extended calculus of constructions, 1990.

[12] Alexandre Miquel. Forcing as a program transformation. In *LICS*, pages 197–206, 2011.

[13] Paulo Oliva. Unifying functional interpretations. *Notre Dame Journal of Formal Logic*, 47(2):263–290, 2006.

[14] Thomas Streicher and Ulrich Kohlenbach. Shoenfield is Gödel after Krivine. *Math. Log. Q.*, 53(2):176–179, 2007.

A formalization of the Quipper quantum programming language

Neil J. Ross

Dalhousie University, Halifax, Canada
neil.jr.ross@dal.ca

Quipper is a programming language for quantum computation ([1], [2]). At the moment, Quipper is implemented as an embedded language, whose host language is Haskell. This means that Quipper can be seen as a collection of predefined Haskell functions and data types, together with a preferred style of writing embedded programs, called an idiom.

Developing a programming language as an embedded language has many advantages. In particular, it allows for the rapid implementation of a large-scale system. This is one of the reasons why Quipper is already a rich language. However, there are also disadvantages to Quipper being an embedded language. One of these is that the Haskell type system, while providing many type-safety properties, is not in general strong enough to ensure full type-safety of the quantum programs. In the current Quipper implementation, it is therefore the programmer's responsibility to ensure that quantum components are plugged together in physically meaningful ways. This means that certain types of programming errors will not be prevented by the compiler. In the worst case, this may lead to ill-formed output or run-time errors.

In this talk, we present a quantum programming language which we call *Proto-Quipper*. This language formalizes a fragment of Quipper and is defined as a typed lambda calculus equipped with a reduction strategy. It is a type-safe language in the sense that it enjoys the usual Subject Reduction and Progress properties. Proto-Quipper provides a foundation for the development of a stand-alone (i.e., non-embedded) version of Quipper. It is designed to "enforce the physics", in the sense that it would detect, at compile-time, programming errors that could lead to ill-formed or undefined circuits.

In designing the Proto-Quipper language, our approach was to start with a limited, but still expressive, fragment of the Quipper language and make it completely type-safe. This fragment will serve as a robust basis for future language extensions. The idea is to eventually close the gap between Proto-Quipper and Quipper by extending Proto-Quipper with one feature at a time while retaining type-safety.

Our main inspiration for the design of Proto-Quipper is the *quantum lambda calculus* of [3]. One of the main differences between Quipper and the quantum lambda calculus is that Quipper acts as a circuit description language. It provides the ability to treat circuits as data, and to manipulate them as a whole. For example, Quipper has operators for reversing circuits, decomposing them into gate sets, etc. By contrast, the quantum lambda calculus only manipulates qubits and all quantum operations are immediately carried out on a quantum device, not stored for symbolic manipulation.

We therefore extend the quantum lambda calculus with the minimal set of features that makes it Quipper-like. The current version of Proto-Quipper is designed to:

- incorporate Quipper's ability to generate and act on quantum circuits, and
- provide a linear type system to guarantee that the produced circuits are physically meaningful (in particular, properties like no-cloning are respected).

To achieve these goals, we extend the types of the quantum lambda calculus with a type $Circ(T, U)$ of circuits, and add constant terms to capture some of Quipper's circuit-level operations, like reversing. The execution of Proto-Quipper programs is formalized as a reduction relation on pairs $[C, t]$ of a term t of the language and a so-called circuit state C. The state C represents the circuit currently being built. The reduction is defined as a rewrite procedure on such pairs, with the state being affected by redexes involving quantum constants.

This is joint work with Henri Chataing and Peter Selinger.

References

[1] A.S. Green, P. Lefanu Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. An introduction to quantum programming in quipper. In *Proceedings of the 5th International Conference on Reversible Computation (RC 2013), Victoria, BC, Canada*, volume 7948, pages 110–124. Springer Lecture Notes in Computer Science, 2013.

[2] A.S. Green, P. Lefanu Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013), Seattle*, volume 48(6), pages 333–342, 2013.

[3] P. Selinger and B. Valiron. Quantum lambda calculus. In S. Gay and I. Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University Press, 2009.

Exceptions in Dependent Type Theory

Jorge Luis Sacchini*

Carnegie Mellon University, Doha, Qatar
sacchini@qatar.cmu.edu

Exceptions provide a convenient mechanism for signaling errors in a program. Signaling is performed by *raising an exception*, which effectively causes a non-local transfer to a dynamically-installed *handler* that can capture the exception and perform some action, e.g. recover from the error situation. If no handler is found, execution is aborted. Most modern programming languages provide built-in mechanisms for raising and handling exceptions.

In dependently-typed programming languages, such as Agda [5], Coq [6], exceptions can be encoded using sum-types. For example, a division function could be given type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} + \mathsf{div\_by\_zero}$. By giving such types a monadic structure, an approach popularized by Haskell, programming with encoded exceptions is rather straightforward. One of the benefits of this approach is that exceptions are encoded in types, which means that a compiler can enforce that all exceptions are handled. Furthermore, in the case of Coq and Agda, logical consistency is preserved, as the language is not changed.

On the other hand, having a primitive notion of exceptions has a number of advantages over encoded exceptions. First, primitive exceptions are more convenient for programming. For example, the expression $1 + \mathsf{div}\,1\,0$ is valid (when executed it would raise an exception), while in the encoding given above, we first have to analyze the value returned by div before proceeding with the addition. This affects performance as well, as we have to pack and unpack the encoded exceptions at every use. Second, exceptions have better support for modularity and code reuse [3]. For example, a higher-order function can be passed as argument a function that may raise exceptions without jeopardizing type safety.

Adding support for first-class exceptions in dependent type theory poses several challenges. First, exceptions usually can have any type; for example 0 and $\mathsf{raise}\,\mathsf{div\_by\_zero}$ can have both type nat. This is undesirable in a dependent type theory as it would lead to logical inconsistencies. To overcome this problem, we need to keep track of exceptions in the type system. We follow the approach given by Lebresne [4]. He proposes a type constructor of the form $A \uplus \psi$, where A is a type and ψ is a set of exceptions. This type is essentially a sum type between regular and exceptional values. Hence, 0 has type nat, while $\mathsf{raise}\,\mathsf{div\_by\_zero}$ has type $\mathsf{nat} \uplus \{\mathsf{div\_by\_zero}\}$.

Second, exception impose a fixed evaluation order—usually, but not always, call-by-value (CBV). In a dependent type theory, where we intent to reason about open terms, we do not want to commit to any evaluation order. To solve this problem, David and Mounier [2] and also Lebresne [4] consider *call-by-name* exceptions. The idea is to have the reduction rules:

$$(\lambda x.t)\,u \to t[u/x] \qquad\qquad (\mathsf{raise}\,\varepsilon)\,u \to \mathsf{raise}\,\varepsilon$$

So that $(\lambda x.t)\,(\mathsf{raise}\,\varepsilon)$ reduces to $t[\mathsf{raise}\,\varepsilon/x]$. This reduction rules, including the usual behavior for handling exceptions, result in a confluent relation.

Third, we mentioned that one advantage of the exception mechanism is better support for modularity and code reuse. The typical example is a sorting function that takes a comparison

*This work was made possible by a NPRP grant (NPRP 09-1107-1-168) from the Qatar National Research Fund (a member of The Qatar Foundation). The statements made herein are solely the responsibility of the author.

function of type, let us say, $\mathsf{nat} \to \mathsf{nat} \to \mathsf{bool}$. If we pass an argument of type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{bool} \uplus \{\varepsilon\}$, we expect that the result will still be well typed, even if it may raise exception ε when applied to a list (i.e. we expect the result to be of type $\mathsf{list}\,\mathsf{nat} \uplus \{\varepsilon\}$). Lebresne [4] introduces *corrupted types* to allow this behavior. A corrupted type has the form A^{ψ}, where ψ is a set of exceptions. A term of type A^{ψ} can be seen as a term of type A where some subexpression is replaced by $\mathsf{raise}\,\varepsilon$ (with $\varepsilon \in \psi$). Corruption has nice properties like distribution across function types: $(A \to B)^{\psi} = A^{\psi} \to B^{\psi}$; this means that a sorting function can have type $(\mathsf{nat}^{\psi} \to \mathsf{nat}^{\psi} \to \mathsf{bool}^{\psi}) \to \mathsf{list}^{\psi}\,\mathsf{nat} \to \mathsf{list}^{\psi}\,\mathsf{nat}$, for any ψ (including $\emptyset$), effectively allowing reuse of the function in the presence of exceptions.

In general, we expect that, in any well-typed program of A, replacing any subexpression by $\mathsf{raise}\,\varepsilon$ would result in a well-typed program of type A^{ψ}. However, this works in simply-typed systems, but not in the presence of dependent types. For example, consider a function P of dependent type

$$\begin{aligned}\Pi x{:}\mathsf{nat}.\ \mathsf{case}\,(\mathsf{try}\ x\ \mathsf{ow}\ \varepsilon \Rightarrow \mathsf{S}\,\mathsf{O})\ \mathsf{of}\\ \mid \mathsf{O} \Rightarrow \mathsf{nat} \to \mathsf{nat}\\ \mid \mathsf{S}\,x \Rightarrow \mathsf{nat}\end{aligned}$$

Then $P\,\mathsf{O}\,\mathsf{O}$ has type nat, but $P\,(\mathsf{raise}\,\varepsilon)\,\mathsf{O}$ is not well typed. Although this example is a bit artificial, it shows that corruption cannot be applied freely in the presence of dependent types.

In this proposed talk, we will present $\lambda^{\Pi,\varepsilon}$, a predicative type theory with inductive types and call-by-name exceptions. The type system features union types of the form $T \uplus \psi$ to account for exceptions at top level, and corrupted types of the form I^{ψ}, for inductive types I, meaning that an exception can occur under constructors. $\lambda^{\Pi,\varepsilon}$ enjoys desirable metatheoretical properties such as subject reduction and strong normalization (proved using a modified Λ-set model [1]).

However, for the reasons explained above, $\lambda^{\Pi,\varepsilon}$ does not feature a full corruption operator. This implies that, although $\lambda^{\Pi,\varepsilon}$ is more convenient to use than encoded exceptions, it still does not have all the advantages of using primitive exceptions. In this talk, we will also discuss the limitations of $\lambda^{\Pi,\varepsilon}$ and possible ways to overcome them.

References

[1] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, November 1993.

[2] R. David and G. Mounier. An intuitionistic Λ-calculus with exceptions. *J. Funct. Program.*, 15(1):33–52, January 2005.

[3] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. In Barbara G. Ryder and Benjamin G. Zorn, editors, *PLDI*, pages 25–36. ACM, 1999.

[4] Sylvain Lebresne. A system F with call-by-name exceptions. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 323–335. Springer, 2008.

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

[6] The Coq Development Team. *The Coq Reference Manual, version 8.3*, 2009. Distributed electronically at `http://coq.inria.fr/doc`.

Deciding unique inhabitants with sums
(work in progress)

Gabriel Scherer

INRIA – Gallium

Our ongoing work focuses on types that have a unique inhabitant—modulo program equivalence. If we were able to detect when such types appear in a program, we could perform *program inference* to releave the programmer from the obligation to write the less-interesting parts of the program. Unicity of inhabitant is a strong form of principality: inference cannot make a “wrong” guess as there is only one choice. This property can be contrasted to two different notions, inhabited types and subtyping coercions, that it respectively refines and extends:

- The usual provability notion of “having at least an inhabitant”, which is the one of concern when using strongly-typed lambda-calculi to prove mathematical facts. In this setting, the dynamic semantics of terms is ignored, which makes it unsuited to *program* inference.
- The common programming notion of erasable subtyping between two types $A \leq B$: inference systems for subtyping can be seen as a restrictive type system for functions whose terms have a computational interpretation which is *always* the identity function—when it is inhabited, $A \leq B$ has a unique inhabitant. On the contrary, some types have unique inhabitants that are not the identity functions, such as `swap :` $\forall AB.\ (A * B) \rightarrow (B * A)$.

To decide uniqueness, we must be able to enumerate the *distinct* terms at a given type. As a first step, we consider the simply-typed lambda-calculus with arrows, product and sums. We are looking for a term enumeration process that is *complete*, i.e., it does not miss any computational behavior, and *canonical*, i.e., it has no duplicates. We propose an approach based on *saturation*, with encouraging results, although termination is still a conjecture.

Computational completeness Some existing proof calculi, such as *contraction-free* calculi [Dyc13], make simplifications designed to make provability decision tractable, but throw away some behaviors, loosing computational completeness. The following rule, which drops a function after its first invocation, preserves provability even in a contraction-free calculus:

$$\frac{\Gamma, A \rightarrow B \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \rightarrow B \vdash C}$$

Consider a datatype $A * (A \rightarrow (A * B))$ of infinite streams of B with internal state A. If the generating function is dropped after its first result, there is exactly one proof of $A * (A \rightarrow (A * B)) \vdash B$ (getting the first element of the stream), while there are infinitely many observably distinct programs at that type.

Focusing Focusing [And92] imposes a phase discipline on derivations by distinguishing *invertible* and *non-invertible* inference rules—invertible rules are those whose inverse is derivable. In absence of sums, focused proofs are in exact correspondance to β-short η-long proof terms; it is computationally complete. Starting from a grammar for values and neutrals:

$$\begin{array}{rcllll} v & ::= & \lambda(x\,{:}\,A)\, v & \mid\ (v, v) & & \mid\ n \\ n & ::= & n\ v & \mid\ \pi_1\, n & \mid\ \pi_2\, n & \mid\ x \end{array}$$

we can define the set $\texttt{Val}(\Gamma \vdash A)$ of (distinct) values of type A in the environment Γ, along with the set $\texttt{Ne}(\Gamma \vdash A)$ of neutrals at this type, by lifting notations from terms to sets of terms:

$$\begin{array}{lcllcl}
\texttt{Val}(\Gamma \vdash A \to B) & := & \lambda(x\!:\!A)\,\texttt{Val}(\Gamma, x\!:\!A \vdash B) & \texttt{Ne}(\Gamma \vdash A_i) & \supseteq & \pi_i\,\texttt{Ne}(\Gamma \vdash A_1 * A_2) \\
\texttt{Val}(\Gamma \vdash A * B) & := & (\texttt{Val}(\Gamma \vdash A),\ \texttt{Val}(\Gamma \vdash B)) & \texttt{Ne}(\Gamma \vdash B) & \supseteq & \texttt{Ne}(\Gamma \vdash A \to B)\,\texttt{Val}(\Gamma \vdash B) \\
\texttt{Val}(\Gamma \vdash X) & := & \texttt{Ne}(\Gamma \vdash X) \quad (X \text{atomic}) & \texttt{Ne}(\Gamma \vdash N) & \supseteq & \{x \mid (x\!:\!N) \in \Gamma\}
\end{array}$$

Note that Val is structurally recursive on the input type; it corresponds to the invertible rules. On the contrary, Ne corresponds to non-invertible rules, and is defined as a least fixpoint: this is where the aforementioned termination control techniques are necessary. The following property is key to showing that this enumeration is canonical:

Lemma 1 (Canonicity of negative neutrals)**.** *For any $n_1, n_2 \in \texttt{Ne}(\Gamma \vdash A)$, if n_1 and n_2 are syntactically distinct, then they are distinct for contextual equivalence.*

This specification can be turned into a decision procedure; termination arguments are of two sorts. First, the *subformula property* gives a finite bound on the number of types that will be considered. Second, cycles in the equations defining Ne (in particular types with an infinite number of distinct inhabitants, such as Church integers $X \to (X \to X) \to X$) can be worked upon using a graph-based representation in the style of Wells and Yakobowski [WY04].

Unfortunately, focusing alone does not capture the notably difficult η-equivalence for sums. Writing $\delta(e_1,\, x.e_2,\, y.e_3)$ for $(\texttt{match}\ e_1\ \texttt{with inl}\,x \to e_2 \mid \texttt{inr}\,y \to e_3)$, the two following terms correspond to distinct, but observationally equivalent, focused proofs of $(1 \to A+B) \to A+B$: $(\lambda(f)\,\delta(f\ 1,\ x.\,\texttt{inl}\,x,\ y.\,\texttt{inr}\,y))$ and $(\lambda(f)\,\delta(f\ 1,\ x.\,\texttt{inl}\,x,\ y.\,\delta(f\ 1,\ x'.\,\texttt{inl}\,x',\ y'.\,\texttt{inr}\,y')))$.

Saturation Coupling an enumeration of focused proofs with an equivalence checking procedure for sums [Lin07] does not work, as there may be infinitely many redundant copies. The main idea of these algorithms is to move sum elimination as high in the term as possible. We thus propose to eliminate sums as early as possible during term generation, by simutaneously eliminating all neutrals of any positive type P when the goal is itself a positive Q:

$$\texttt{Val}(\Gamma, x\!:\!(A+B) \vdash C) := \delta(x,\ y.\texttt{Val}(\Gamma, y\!:\!A \vdash C),\ z.\texttt{Val}(\Gamma, z\!:\!B \vdash C))$$
$$\texttt{Ne}(\Gamma \vdash A_1 + A_2) \supseteq_{i \in \{1,2\}} \sigma_i\ \texttt{Ne}(\Gamma \vdash A_i)$$
$$\texttt{Val}(\Gamma \vdash Q) := \texttt{Ne}(\Gamma \vdash Q) \cup (\texttt{let}\ \Delta = (\cup_P\,\texttt{Ne}(\Gamma \vdash P))\ \texttt{in}\ \texttt{Val}(\Gamma, \Delta \vdash Q))$$

This saturation process is complete and canonical. Remarkably, it is strongly related to the notion of “maximal multi-focusing” [CMS08]; our proofs are also maximal in this sense.

While completeness and non-duplicability are relatively simple, termination is still a conjecture. In addition to the “inner” non-termination related to the definition of $\texttt{Ne}(\Gamma \vdash A)$ (infinite number of distinct terms), there is now an “outer” termination problem of infinite alternance of Val and Ne layers. Previous approaches gave either completeness and termination (focusing, with duplicates) or canonicity and termination (incomplete provability calculi), while this presentation is naturally complete and canonical, but not yet proved terminating.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[CMS08] Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *IFIP TCS*, pages 383–396, 2008.

[Dyc13] Roy Dyckhoff. Intuitionistic decision procedures since gentzen. In *Advances in Proof Theory*, 2013.

[Lin07] Sam Lindley. Extensional rewriting with sums. In *TLCA*, pages 255–271, 2007.

[WY04] J. B. Wells and Boris Yakobowski. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR*, pages 262–277, 2004.

On the Complexity of Negative Quantification*

Aleksy Schubert, Paweł Urzyczyn,[1] and Konrad Zdanowski[2]

[1] University of Warsaw
[alx,urzy]@mimuw.edu.pl
[2] Cardinal Stefan Wyszyński University in Warsaw
k.zdanowski@uksw.edu.pl

Abstract

Universal quantifiers in first-order formulas are classified as positive (co-variant) and negative (contra-variant) depending on their position. We define the negative fragment of a logic so that it consists of formulas where all quantifiers are negative. We prove that the decision problem for the negative forall-arrow fragment of intuitionistic predicate logic becomes *co*-NEXPTIME complete under the restriction that all predicates are of fixed arity. This contrasts with an earlier result that the full negative logic is EXPSPACE complete.

The usual way of classifying the quantifier complexity of a formula is by counting the number of quantifier alternations. This concept can be defined even if formulas of a given logic have no equivalent prenex normal form. One simply counts alternations between positive and negative occurrences of quantifiers. In the context of intuitionistic logic this gives the, so called, Mints hierarchy [1]. Here, we are interested in the first level of this hierarchy consisting of formulas with only negative occurrences of quantifiers. For the logic with universal quantification and implication only this class may be defined by the following grammar, where quantifier free formulas are represented by Δ:

- $\Sigma_1 ::= \Delta \mid \Pi_1 \to \Sigma_1$;
- $\Pi_1 ::= \Delta \mid \Sigma_1 \to \Pi_1 \mid \forall x\, \Pi_1$.

We establish the complexity of the set of Σ_1 theorems of intuitionistic predicate logic in a vocabulary with predicates of bounded arity and with no function symbols. We show that this fragment is *co*-NEXPTIME complete. This may be compared with the earlier work which shows that Σ_1 fragment with no restriction on the arity of predicates is EXPSPACE complete [3], as well as with Π_1 fragment which is 2-*co*-NEXPTIME hard [2]. The higher classes in the hierarchy, Σ_2 and Π_2, become undecidable even in the monadic vocabulary [3].

To obtain a lower bound we define a tiling problem of covering the space $\mathbb{N} \times \{0,1\}^*$ with a finite set of tiles $\mathcal{T}$, according to a given set of deterministic rules $\mathcal{G} : \mathcal{T}^4 \to \mathcal{T}^2$. The set of tiles $\mathcal{T}$ is assumed to include two distinguished elements, E and OK. Rules in $\mathcal{G}$ define a tiling function $T_{\mathcal{G}} : \mathbb{N} \times \{0,1\}^* \to \mathcal{T}$, given by:

- $T_{\mathcal{G}}(n, w) = \mathrm{E}$, when $n = 0$ or $w = \varepsilon$.
- $T_{\mathcal{G}}(m{+}1, wi) = \pi_i(\mathcal{G}(\mathrm{K}, \mathrm{L}, \mathrm{M}, \mathrm{N}))$, for $i = 0, 1$, where π_i stands for the i-th projection, and $\mathrm{K} = T_{\mathcal{G}}(m, wi)$, $\mathrm{L} = T_{\mathcal{G}}(m, w)$, $\mathrm{M} = T_{\mathcal{G}}(m{+}1, w)$, and $\mathrm{N} = T_{\mathcal{G}}(m{+}2, w)$;

One can imagine a tiling of $\mathbb{N} \times \{0,1\}^*$ as a full binary tree labeled by rows of tiles (the label of a node $w \in \{0,1\}^*$ is the sequence of tiles $T_{\mathcal{G}}(n, w)$, for all n).

*Project supported through NCN grant DEC-2012/07/B/ST6/01532.

Let $s \in \mathbb{N}$. The problem $\mathcal{G}$ is *s-solvable* iff, for every w with $|w| = s$, there is a prefix w' of w and a number $m \leq s$ such that $T_{\mathcal{G}}(m, w') = \text{OK}$. That is, an OK tile must be reached at every branch of the tree of length s and it must be at most the s-th tile in the row.

Since s is presented in binary, one can show by a routine argument that the problem whether a given $\mathcal{G}$ is s-solvable belongs to the class of *co*-NEXPTIME complete problems. We show how to code this problem by a Σ_1 formula φ such that s–solvability of $\mathcal{G}$ corresponds to provability of φ. Moreover, φ can be written as $\forall \bar{y}_1 \psi_1 \to \cdots \to \forall \bar{y}_k \psi_k \to \psi_{k+1}$, where all ψ_i are quantifier free. After encoding s-solvability of $\mathcal{G}$ using predicates of arity 2, we use a syntactic translation to replace them by monadic predicates.

For the upper bound we define a refutation system for the Σ_1 fragment of intuitionistic logic. We show that the size of a refutation may be bounded by 2^{n^k}, where n is the length of a disproved formula φ and k corresponds, roughly, to the maximal arity of predicates in φ.

The main ingredient here is the observation that while proving a Σ_1 formula φ from the set of Π_1 assumptions Γ there is no need of introducing new variables besides those which are free in φ and Γ (assuming that there is at least one). The above restricts the number of possible sequents that may occur in an alleged proof of $\Gamma \vdash \varphi$. After constructing a refutation tree one can show that some of its branches can be cut off so that the existence of the full refutation tree follows from the existence of its fragment of exponential size.

References

[1] 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.

[2] Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz. Positive logic is 2-EXPTIME hard. In *Types 2013*, pages 72–73. 2013.

[3] Aleksy Schubert, Paweł Urzyczyn, and Konrad Zdanowski. On the Mints hierarchy in first-order intuitionistic logic, 2014. Submitted.

Higher Inductive Types as Homotopy-Initial Algebras

Kristina Sojakova

Carnegie Mellon University, Pittsburgh, USA
kristinas@cmu.edu

Homotopy Type Theory (HoTT, [9]) is a new field of mathematics based on the recently-discovered correspondence between Martin-Löfs constructive type theory and abstract homotopy theory. Under this new interpretation, types are topological spaces, terms are points in spaces, and proofs of identity are paths between points. Since proofs of identity are themselves terms (of an identity type), we can also talk about *higher paths*, for instance, proofs that two proofs of identity are equal, and so on.

Type-theoretically, HoTT is an extension of the intensional Martin-Löf dependent type theory, with two new features motivated by abstract homotopy theory: Voevodsky's *univalence axiom* [3] and *higher inductive types* ([5, 6]). The theory remains intensional in the sense that no form of Streicher's K-rule [8] or the identity reflection rule are admissible (as the groupoid model constructed in [2] shows). In fact, the former is incompatible with univalence.

Higher inductive types are important because they allow us to represent a variety of mathematical objects - such as spheres, tori, pushouts, and quotients - within the type theory. Ordinarily, an inductive type X can be understood as being freely generated by a collection of constructors for X. A higher inductive types X also permits constructors involving *path spaces* of X: for example, the circle S^1 can be represented as the higher inductive type generated by a single point constructor $\mathsf{base} : \mathsf{S}^1$ and a single path constructor $\mathsf{loop} : \mathsf{base} =_{\mathsf{S}^1} \mathsf{base}$, where we use $=$ to denote propositional equality. The induction principle associated with this simple definition is powerful enough to show, e.g., that the fundamental group of S^1 is the group of integers (see [4]).

We investigate a variant of higher inductive types whose computational behavior is determined up to a higher path. We show that in this setting, higher inductive types are characterized by the universal property of being a homotopy-initial algebra. In the case of the circle S^1, the data $(\mathbf{S}^1, \mathsf{base}, \mathsf{loop})$ together can be thought of as defining an $\mathbf{S}^1$*-algebra.* The recursion principle for the circle then says that given any other $\mathbf{S}^1$-algebra (X, x, s), where X is a type, $x : X$ is a point, and s is a loop based at x, there exist an $\mathbf{S}^1$*-homomorphism* (f, β, θ) from $(\mathbf{S}^1, \mathsf{base}, \mathsf{loop})$ to (X, x, s).

An $\mathbf{S}^1$-homomorphism (f, β, θ) between $\mathbf{S}^1$-algebras (X, x, s) and (Y, y, r) consists of a map $f : X \to Y$, a path $\beta : f(x) = y$, and a higher path $\theta : f(s) \cdot \beta = \beta \cdot r$, where $f(s)$ records the effect of the map f on the path s. An $\mathbf{S}^1$-algebra is called *homotopy-initial* [1] if the type of homomorphisms to any other algebra is contractible, meaning it consists of an element which is unique up to a higher path, which is itself unique up to a higher path, and so on.

Theorem 1. *An $\mathbf{S}^1$-algebra satisfies the formation, introduction, elimination, and computation rules for a circle if and only if it is homotopy-initial.*

This theorem follows from an analogous result for a more general class of higher inductive types we call *W-suspensions*. For the full account we refer to [7].

References

[1] S. Awodey, N. Gambino, and K. Sojakova. Inductive types in Homotopy Type Theory. In *Logic in Computer Science (LICS 2012)*, pages 95–104. IEEE Computer Society, 2012.

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

[3] C. Kapulkin, P. Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. Available at arxiv.org as arXiv:1211.2851v1, 2012.

[4] D. Licata and M. Shulman. Calculating the fundamental group of the circle in Homotopy Type Theory. In *Logic in Computer Science (LICS 2013)*, pages 223–232. IEEE Computer Society, 2013.

[5] P. Lumsdaine. Higher inductive types: a tour of the menagerie, 2011. Post on the Homotopy Type Theory blog. `http://homotopytypetheory.org/2011/04/24/higher-inductive-types-a-tour-of-the-menagerie/`.

[6] M. Shulman. Homotopy Type Theory, VI, 2011. Post on the n-category cafe blog. `http://golem.ph.utexas.edu/category/2011/04/homotopy_type_theory_vi.html`.

[7] K. Sojakova. Higher inductive types as homotopy-initial algebras. Technical Report CMU-CS-14-101, Carnegie Mellon University, 2014. Available at `http://reports-archive.adm.cs.cmu.edu/`.

[8] T. Streicher. Investigations into intensional type theory, 1993. Habilitation Thesis. Available from the authors web page.

[9] The Univalent Foundations Program, Institute for Advanced Study. *Homotopy Type Theory - Univalent Foundations of Mathematics*. Univalent Foundations Project, 2013.

Proof-relevant rewriting strategies in Coq

Matthieu Sozeau

Inria Paris & PPS, Univ Paris Diderot (UMR-CNRS 7126)
matthieu.sozeau@inria.fr

Abstract

We introduce various enhancements of the generalized rewriting system of Coq. First, we show how the framework can be lifted to proof-relevant relations using the newly introduced universe polymorphic definitions in Coq. Second, we introduce rewriting strategies as a monadic combinator library on top of this framework, resembling the LogicT monad for proof-search (i.e., with backtracking, well-behaved choice and composition). These new features combine to provide a general tool for fine-tuned automated rewriting applicable not only to propositional relations but also general computational type-valued relations. Last, we will also present an idea to handle dependent rewriting, e.g. the ability to rewrite in the domain of a dependent product and get corresponding transportation obligations.

1 Proof-relevant rewriting

The new universe polymorphic extension of Coq [1] is crucial to allow a straightforward generalization of the generalized rewriting framework in Coq [2]. Indeed, the basic notion of proper morphism was previously defined only on propositional relations:

```
Class Proper {A : Type} (R : A → A → Prop) (x : A) :=
  proper_prf : R x x.
```

Although this definition could be written with a Type-valued relation instead in previous versions, generic lemmas about Proper would immediately fix the universe level and would force all relations for which rewriting was used to live at the same level, i.e., a no-go. With universe polymorphism, Proper can be made polymorphic on the $\mathtt{Type}_l$ codomain of the relation and similarly generic lemmas can be instantiated at arbitrary, potentially unrelated levels (for example, Proper itself can be shown to be proper for equivalent relations). Examples of useful proof-relevant rewriting relations abound: the appartness predicate of the reals in the CoRN library, the paths relation of Homotopy Type Theory, and in general, Hom-types of categories are naturally formalized using Type-valued relations.

2 Rewriting strategies

The generalized rewriting tactic is based on a monolithic function that folds through a term and produces type-class constraints for showing that constants are morphisms and applies the rewriting at the appropriate places.

We propose a generalization of this design based on a set of strategies that can be combined to obtain custom rewriting procedures. Its set of strategies is based on Elan's rewriting strategies [3]. Rewriting strategies are applied using the tactic `rewrite_strat` s where s is a strategy expression (already part of Coq 8.4, although undocumented). Strategies are defined inductively as described by the grammar given in Figure 1. Actually a few of these are defined in term of the others using a primitive fixpoint operator:

$$\begin{array}{lcl}
\mathtt{try}\ s & = & s \mathbin{||} \mathtt{id} \\
\mathtt{any}\ s & = & \mathtt{fix}\ u.\mathtt{try}\ (s\ ;\ u) \\
\mathtt{repeat}\ s & = & s\ ;\ \mathtt{any}\ s \\
\mathtt{bottomup}\ s & = & \mathtt{fix}\ bu.((\mathtt{progress}\ (\mathtt{subterms}\ bu)) \mathbin{||} s)\ ;\ \mathtt{try}\ bu \\
\mathtt{topdown}\ s & = & \mathtt{fix}\ td.(s \mathbin{||} (\mathtt{progress}\ (\mathtt{subterms}\ td)))\ ;\ \mathtt{try}\ td \\
\mathtt{innermost}\ s & = & \mathtt{fix}\ i.((\mathtt{subterm}\ i) \mathbin{||} s) \\
\mathtt{outermost}\ s & = & \mathtt{fix}\ o.(s \mathbin{||} (\mathtt{subterm}\ o))
\end{array}$$

s, t, u	::=	`(<-)?` c	(right to left?) lemma
	|	`fail` | `id`	failure | identity
	|	`refl`	reflexivity
	|	`progress` s	progress
	|	`try` s	failure catch
	|	s `;` u	composition
	|	s || t	left-biased choice
	|	`repeat` s	iteration (+)
	|	`any` s	iteration (*)
	|	`subterm(s)?` s	one or all subterms
	|	`innermost` s	innermost first
	|	`outermost` s	outermost first
	|	`bottomup` s	bottom-up
	|	`topdown` s	top-down
	|	`hints` *hintdb*	apply first matching hint
	|	`terms` $c \dots c$	any of the terms
	|	`eval` *redexpr*	apply reduction
	|	`fold` c	fold expression
	|	`pattern` p	pattern matching

Figure 1: Rewriting strategy syntax

Apart from the basic control strategies, we have lemmas strategies allowing to apply any of a set of rewrite rules (`hints`, `terms`), an evaluation strategy that applies anywhere and reduces according to a selection of the usual $\beta\delta\iota\zeta$-laws of CIC, and a `fold` strategy that can refold constants up to unification. The special `pattern` strategy succeeds only when the term pattern-matches its argument, allowing selective rewriting. With these combinators, we can subsume a few existing tactics of Coq: variants of `rewrite`, `autorewrite`, `eval`/`unfold` and `fold`, in the generalized rewriting setting. We also allow user-defined, fine-tuned strategies, whose performance improves on the `repeat rewrite` strategy implemented by the `autorewrite` tactic for example.

On the implementation side, these strategies are implemented using a success-failure continuation monad, similar to the `LogicT` monad [4], which has efficient backtracking and clear semantics while keeping the code modular. In particular, the combination with universe polymorphism was easy to add thanks to this modularity and the hiding of state passing which is required when manipulating polymorphic constants in the ML tactic.

3 Conclusion

These new features make generalized rewriting a viable framework to work with proof-relevant relations, in particular the `paths` type of Homotopy Type Theory, and other proof-relevant notions of equality like isomorphisms of types or appartness in real number formalizations. We hope to apply the enhanced tactic to real-world formalizations in the future.

References

[1] Sozeau, M., Tabareau, N.: Universe Polymorphism in Coq. In: ITP'14. (2014) To appear.

[2] Sozeau, M.: A New Look at Generalized Rewriting in Type Theory. Journal of Formalized Reasoning **2**(1) (December 2009) 41–62

[3] Luttik, S.P., Visser, E.: Specification of Rewriting Strategies. In: 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing, Springer-Verlag (1997)

[4] Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers: (functional pearl). In: ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2005) 192–203

Towards an Internalization of the Groupoid Interpretation of Type Theory

Matthieu Sozeau[1] and Nicolas Tabareau[2]

[1] Inria πr^2, Preuves, Programmes et Systèmes (PPS)
[2] Inria Ascola, Laboratoire d'Informatique de Nantes Altantique (LINA)
`firstname.surname@inria.fr`

Abstract

Homotopical interpretations of Martin-Löf type theory lead toward an interpretation of equality as a richer, more extensional notion. Extensional or axiomatic presentations of the theory with principles based on such models do not yet fully benefit from the power of dependent type theory, that is its computational character. Reconciling intensional type theory with this richer notion of equality requires to move to higher-dimensional structures where equality reasoning is explicit. In this paper, we follow this idea and develop an internalization of a groupoid interpretation of Martin-Löf type theory with one universe respecting the invariance by isomorphism principle. Our formal development relies crucially on ad-hoc polymorphism to overload notions of equality and on a conservative extension of Coq's universe mechanism with polymorphism.

1 Introduction

Our work here concentrates on the internalization in Coq of Hofmann and Streicher's groupoid model where we can have a self-contained definition of the structures involved.

Our first motivation to implement this translation is to explore the interpretation of type theory in groupoids in a completely intensional setting and in the type theoretic language, leaving no space for imprecision on the notions of equality and coherence involved. We also hope to give with this translation a basic exposition of the possible type theoretic implications of the groupoid/homotopy models, bridging a gap in the literature. On the technical side, the definition of the groupoid model actually requires to reason at a 2-dimensional level. This is due to the way we interpret the strictness in the definition of groupoids. Indeed, interpreting strictness by the fact that the internal equality coincides with the identity type requires the functional extensionlaty axiom when it comes to define for instance the groupoid on the function space. Our interpretation of strictness is closer to the idea that a groupoid is a weak ω-groupoid for which all equalities at dimension 2 are the same. That is, we only use identity types to express triviality of higher dimension, not coherences themselves. As the model that we use does not have the uniqueness of identity proof principle, the two ways of formalizing groupoids mentionned above are different. Our presentation requires less properties on identity types, but we still need the axiom of functional extensionality. Also, this indicates that if we scale to ω-groupoids, the presence of identity types in the core type theory will not be necessary anymore and so the core type theory will be axiom free. Thus, our work can be seen as a proof of concept that it is possible to interpret homotopy type theory into type theory without identity types.

We use an extension of the Coq proof assistant [1] with universe polymorphism to formally define our translation [2]. We studied a restricted source theory resembling a cut-down version of the core language of the Coq system, with only one `Type` universe (see [3] for an in-depth study of this system). This is basically Martin-Löf Type Theory (without `Type : Type`), with Π, Σ, `Id` types and a single universe $\mathcal{U}$.

Universe and Type Equivalence. The universe $\mathcal{U}$ is closed under Σ, Π, $\mathbb{0}$, $\mathbb{1}$, $\mathbb{2}$ and `Id` in elements of $\mathcal{U}$, *not* type equivalences. For T and U in $\mathcal{U}$, the type of (Set)-isomorphisms $T \equiv U$, is definable directly using the other type constructors. The new, proof-relevant type equivalence in the source theory for which we want to give a computational model appears in the rule below, extending the formation rules of identity types on $\mathcal{U}$. The `J` rule for type equivalences witnesses the invariance under isomorphism principle of the source type theory.

$$\frac{\Gamma \vdash i : \texttt{Elt}(A) \equiv \texttt{Elt}(B)}{\Gamma \vdash \texttt{equiv}\ i : \texttt{Id}_{\mathcal{U}}\ A\ B}$$

2 Formalization of groupoids

We formalize groupoids using type classes. Contrarily to what is done in the setoid translation, the basic notion of a morphism is an inhabitant of a relation on a type T in Type (i.e., a proof-relevant relation).

In our definition of the type Type_1 of groupoids, we do not ask that the internal equality coincides with the identity type but we model explicitly coherence laws with an equality at dimension 2, which is assumed to be irrelevant. This irrelevance is defined using a notion of contractibility expressed with identity types. One can then define groupoid morphisms (functors) preserving homs which form a pre-category with natural transformations and modifications. Groupoid equivalence itself is formalized using adjoint equivalences. We can define the pre-groupoid Type_1^1 of groupoids and homotopy equivalences. However, groupoids together with homotopy equivalences do not form a groupoid but rather a 2-groupoid. As we only have a formalization of groupoids, this can not be expressed in our setting. Nevertheless, we can state that setoids (inhabitants of Type_0, which are the targets in the interpretation of the types of our universe $\mathcal{U}$) form a groupoid. The proof that it is indeed a groupoid makes use of functional extensionality to prove contractibility of higher cells. As Type_1 appears both in the term and in the type, the use of polymorphic universe is crucial here to avoid an inconsistency.

3 Interpretation of the source type theory

Our formalization of groupoids can be organized into a model of dependent type theory. The interpretation is based on the notion of categories with families introduced by Dybjer [4] later used in [5]. This interpretation can also be seen as an extension of the Takeuti-Gandy interpretation of simple type theory, recently generalized to dependent type theory by Coquand et al. using Kan semisimplicial sets or cubical sets [6]. In our interpretation, we take advantage of universe polymorphism to interpret dependent types directly as functors into Type_0^1. We interpret contexts as groupoids. The empty context being the groupoid with exactly one element at each dimension. Types in a context Γ are (context) functors from Γ to the groupoid of setoids Type_0^1. Thus, a judgment $\Gamma \vdash A : \mathsf{Type}$ is represented as a term A of type $\mathsf{Typ}\ \Gamma$. Context extension (Rule DECL) is given by dependent sums, i.e., the judgment $\Gamma, x : A \vdash$ is represented as $\Sigma\ A$. Substitution and all typing rules can be interpreted this way. For conversion, we just have (trivial) metatheoretical result that we preserve conversion in the interpretation, as the intepretation of types just adds compatibility terms to a type, so two convertible types in the source language just get interpreted as two pairs with convertible first projections in the shallow embedding.

4 Related Work

The groupoid interpretation is due to Hofmann and Streicher [5]. This interpretation is based on the notion of categories with families introduced by Dybjer [4]. This framework has recently been used by Coquand et al. to give an interpretation in semi-simplicial sets and cubical sets [6, 7]. Although very promising, the interpretation based on cubical sets has not yet been mechanically checked, only an Haskell implementation exists. Observational Type Theory (OTT) [8], an intentional type theory where functional extensionality is native, but equality in the universe is structural.

References

[1] The Coq development team: Coq 8.4 Reference Manual. Inria. (2012)

[2] Sozeau, M., Tabareau, N.: Universe Polymorphism in Coq. In: ITP'14. (2014) To appear.

[3] Hoffman, M.: Syntax and Semantics of Dependent Types. In: Semantics and Logics of Computation. (1997)

[4] Dybjer, P.: Internal type theory. In : Types for Proofs and Programs. LNCS 1158 (1996)

[5] Hofmann, M., Streicher, T.: The Groupoid Interpretation of Type Theory. In: Twenty-five years of constructive type theory (Venice, 1995). Volume 36 of Oxford Logic Guides. Oxford Univ. Press (1998) 83–111

[6] Barras, B., Coquand, T., Huber, S.: A Generalization of Takeuti-Gandy Interpretation. (2013)

[7] Bezem, M., Coquand, T., Huber, S.: A Model of Type Theory in Cubical Sets. (December 2013)

[8] Altenkirch, T., McBride, C., Swierstra, W.: Observational Equality, Now! In: PLPV'07, Freiburg, Germany (2007)

Type system for automated generation of reversible circuits (abstract)

Benoît Valiron

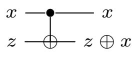
PPS, UMR 7126, Université Paris Diderot,
Sorbonne Paris Cité, F-75205 Paris, France,
benoit.valiron@monoidal.net

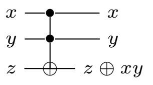
Summary of the abstract

Reversible circuits are boolean circuits made of reversible elementary gates. Such circuits are particularly useful in the context of quantum computation as quantum oracles. A quantum oracle is the circuit representation of a conventional, classical function. In this abstract, we aim at a simple method for generating reversible circuits out of a higher-order functional programming language. We expose the drawbacks of this approach and discuss how a type system could solve some of these problems.

Introduction. Reversible circuits are special kinds of boolean circuits, where wires are horizontal and parallel, and where elementary operations (called *gates*) are reversible. Wires host bits flowing from left to right, all at the same pace. When some bits meet a gate, the corresponding operation is applied on the wires attached to the gate. The overall circuit is reversible by making the bits flow backward.

The gates we consider here are multi-controlled-not gates. A not gate is simply $-\!\oplus\!-$. The corresponding operation is the flip of the value flowing in the wire. The controlled-not gate, also denoted CNOT, is the first gate on the right, while the doubly-controlled gate, also denoted Toffoli is the second. A controlled-not flips the bit only when the controlling wires all have value "true" (or 1).

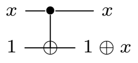


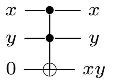


The main use we have in mind for reversible circuits in this abstract is quantum computation. In quantum computation, reversible circuits are mostly used as oracle: the *description* of the problem to solve. Usually, this description is given as a classical, conventional algorithm. This abstract is concerned with the design of reversible circuits derived from such algorithms, when given as programs in a PCF-like language: the language we aim at is minimal but expressive enough to encode most algorithms found in quantum oracles. It should feature recursion, pairs, booleans and lists, and easily be extended with additional structures if needed. We are therefore interested in the *compilation* of a program into a reversible circuit. In this abstract, we present the type system for a small subset of the language, nonetheless enough to discuss an example of term where optimization can happen.

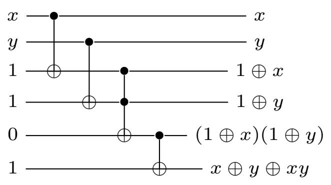
Reversible circuits from programs. Compiling a program into a reversible circuit is fundamentally different from compiling to a regular back-end: there is no notion of "loop", no real control flow, and all branches will be explored during the execution. In essence, a reversible circuit is the *trace* of all possible executions of a given program, and the compilation of a program essentially amounts to its evaluation.

As described by Landauer and Bennett, a conventional, classical algorithm computing a boolean function $f : \texttt{bit}^n \to \texttt{bit}^m$ can be mechanically transformed into a reversible circuit sending the triplet $(x, \vec{0}, \vec{0})$ to $(x, \mathrm{trace}, f(x))$. The input wires are not modified by the circuit, and the trace of all intermediate results are kept in garbage wires. For example, the Landauer embeddings for negation and conjunction are respectively shown in the right: they are simply CNOT and Toffoli gates, together with some auxiliary wires initialized to some constant boolean value 0 or 1.





Because of their particular structure, the Landauer embeddings of functions g and h can be composed to give a Landauer embedding of the composition $h \circ g$: the trace of the execution of both g and h are kept in garbage wires. For example, one can construct the embedding of the disjunction as the composition of the embeddings for the negation and the conjunction, as shown on the right. The disjunction of x and y is $\neg(\neg x \wedge \neg y)$. The Landauer embeddings of negation and conjunction are composed in order to build all intermediary results, until we reach $x \oplus y \oplus xy$, which is indeed $x \vee y$. The two first CNOTs compute the inner negations, the Toffoli computes the conjunction, while the last CNOT makes the outer negation. As we said above, all intermediary results are retained.



One can use this technique to design the compilation of any lambda-calculus manipulating booleans into a reversible circuit. Therefore, if `not` and `and` have their obvious meanings, the two programs

$$x, y : \texttt{bit} \vdash \texttt{not}(\texttt{and}(\texttt{not}\, x)(\texttt{not}\, y)) : \texttt{bit} \tag{1}$$

$$x, y : \texttt{bit} \vdash (((\lambda z.\lambda t.\lambda s.s\,(\texttt{and}\, t\, z))(\texttt{not}\, x))(\texttt{not}\, y))\,\texttt{not} : \texttt{bit} \tag{2}$$

will generate the same circuit for disjunction shown above, regardless of the lambda abstractions involved in the term of Eq. (2).

Circuit optimization and type system. In the naive implementation of the disjunction in Eq. (1), the intermediate result of `and` cannot be reused later on as it is not stored in a variable. So instead of creating a new fresh variable, the outer `not` could simply reuse the wire of the intermediate result of `and` and update it with a $\oplus$ gate. In Eq. (1), this optimization can be realized at circuit-generation time by simple subterm inspection. In Eq. (2), it is not so clear: we claim that a specific type system is the right tool to untangle the terms and decide whether a particular wire is used several times. Terms can then be annotated with type information that is used at circuit-generation time to decide whether a fresh wire should be created or not.

Formally, the language consists of terms, types and sorts. For illustration, in this abstract we only give the subset needed to type Eqs (1) and (2). Terms are $M, N ::= x \mid \lambda x.M \mid MN \mid \texttt{and} \mid \texttt{not}$, types are $A, B ::= \alpha \mid A \to B$, while sorts are $\tau ::= 0 \mid 1 \mid +$. Sorts come with a transitive relation $<$ defined by $0 < 0 < 1 < + < +$. The type α ranges over an infinite alphabet, and it stands for a bit wire. So with respect to Eqs (1) and (2), we essentially have a type `bit` annotated with wires. A wire α of sort 0 is intuitively not used. A wire of sort 1 is used only once in the generated circuit while a wire of sort $+$ is used more than once. A typing judgement is of the form

$$\alpha_1 : \tau_1 \ldots \alpha_n : \tau_n \mid x_1 : A_1 \ldots x_m : A_m \vdash M : B,$$

and the typing rules enforce the property that wires can be shared amongst several variables but are themselves linear. If $|X|$ stands for the set of wires in X, then

$$\frac{|\Delta| = |A|}{\Delta \mid x : A \vdash x : A} \qquad \frac{\Delta \mid \Gamma, x : A \vdash M : B}{\Delta \mid \Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma_1 \mid \Delta \vdash N : A \quad \Gamma_2 \mid \Delta \vdash M : A \multimap B}{\Gamma_1 \cup \Gamma_2 \mid \Delta \vdash MN : B}$$

Finally, constants are typed with $\alpha : \tau_1, \beta : \tau_2 \vdash \texttt{not} : \alpha \to \beta$ when $\tau_1 \geq 1$, and $\alpha : \tau_1, \beta : \tau_2, \gamma : \tau_3 \vdash \texttt{and} : \alpha \to \beta \to \gamma$ when $\tau_1, \tau_2 \geq 1$. In the typing rule for application appears a union of contexts of sorts: $(\alpha_1 : \tau_1 \ldots \alpha_n : \tau_n, \beta_1 : \sigma_1 \ldots \beta_m : \sigma_m) \cup (\alpha_1 : \tau'_1 \ldots \alpha_n : \tau'_n, \beta_{m+1} : \sigma_{m+1} \ldots \beta_k : \sigma_k)$ is equal to $(\alpha_1 : \max(\tau_1, \tau'_1) \ldots \alpha_n : \max(\tau_n, \tau'_n), \beta_1 : \sigma_1 \ldots \beta_k : \sigma_k)$ where $\max(\tau, \tau')$ is the smallest value σ such that $\tau, \tau' \leq \sigma$ and σ is strictly greater than one of τ and τ'. Since $(<)$ is reflexive on sort 0, if a wire α is neither "used" in M nor in N, then it is not used in MN. Again, $(<)$ is not reflexive on sort 1: a wire α used once both in M and N is used "many" times (aka "$+$") on MN.

Thus, during the compilation of a term, every `not` whose first argument is of sort 1 in the final term (i.e. the bottom of the typing derivation) can safely be compiled into a NOT gate, not generating a fresh wire. In the case of Eqs (1) and (2), one can type both terms with the pair of context and type $\ldots, \alpha : 1, \beta : 1, \gamma : 0 \mid x : \alpha, y : \beta \vdash - : \gamma$ and the outer `not` has an argument of sort 1 in both cases, as desired.

Author Index