24th International Conference on
Types for Proofs and Programs

TYPES 2018

Braga, Portugal, 18 June - 21 June 2018

Abstracts

José Espírito Santo and Luís Pinto (eds.)

Centro de Matemática
University of Minho

Braga, 2018

Preface

This volume contains the abstracts of the talks presented at the 24th International Conference on Types for Proofs and Programs, TYPES 2018, to take place in Braga, Portugal, 18 - 21 June 2018.

The TYPES meetings are a forum to present new and on-going work in all aspects of type theory and its applications, especially in formalised and computer assisted reasoning and computer programming. The meetings from 1990 to 2008 were annual workshops of a sequence of five EU funded networking projects. Since 2009, TYPES has been run as an independent conference series, funded by COST Action EUTypes since 2016.

Previous TYPES meetings were held in Antibes (1990), Edinburgh (1991), Båastad (1992), Nijmegen (1993), Båastad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal near Nijmegen (2002), Torino (2003), Jouy-en-Josas near Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Torino (2008), Aussois (2009), Warsaw (2010), Bergen (2011), Toulouse (2013), Paris (2014), Tallinn (2015), Novi Sad (2016), Budapest (2017).

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

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 A formal post-proceedings volume is prepared after the conference; papers submitted to that must represent unpublished work and are subjected to a full review process.

The programme of TYPES 2018 includes four invited talks by Cédric Fournet (Microsoft Research, UK), Delia Kesner (IRIF CNRS and Université Paris-Diderot, France), Matthieu Sozeau (INRIA, France), and Josef Urban (CIIRC, Czech Republic). The contributed part of the programme consists of 42 talks. One of the sessions of the programme pays tribute to Martin Hofmann, and includes three of the contributes talks, and an invited talk by Ralph Matthes (CNRS, IRIT, University of Toulouse, France).

Similarly to the 2011 and the 2013-2017 editions of the conference, the post-proceedings of TYPES 2018 will appear in Dagstuhl's Leibniz International Proceedings in Informatics (LIPIcs) series.

We are grateful for the support of COST Action CA15123 EUTypes, Centro de Matemática da Universidade do Minho, Conselho Cultural da Universidade do Minho, and Câmara Municipal de Braga.

We acknowledge the following people for their support in the organization of TYPES 2018: Fernanda Barbosa, Maria Antónia Forjaz, Cândida Marcelino, Nuno Oliveira, Alexandra Pereira, Miguel Ayres de Campos Tovar, and Maria Francisca Xavier. We express our special gratitude to our colleagues in the organizing committee: Cláudia Mendes Araújo and Maria João Frade.

Braga, June 14, 2018

José Espírito Santo and Luís Pinto

Organization

Program Committee

Andreas Abel	Chalmers University Gothenburg
Amal Ahmed	Northeastern Universtiy
Andrej Bauer	University of Ljubljana
Marc Bezem	University of Bergen
Maria Paola Bonacina	Universita‘ degli Studi di Verona
Ugo De'Liguoro	University of Torino
Gilles Dowek	INRIA and ENS Paris-Saclay
Peter Dybjer	Chalmers University Gothenburg
José Espírito Santo	University of Minho
Herman Geuvers	Radboud University
Ambrus Kaposi	Eötvös Loránd University
Assia Mahboubi	INRIA
Ralph Matthes	IRIT (C.N.R.S. and University of Toulouse III)
Keiko Nakata	SAP Potsdam
Luís Pinto	University of Minho
Andrew Pitts	University of Cambridge
Pierre-Marie Pédrot	Max Planck Institute for Software Systems Saarbruckën
Aleksy Schubert	University of Warsaw
Carsten Schürmann	IT University of Copenhagen
Anton Setzer	University of Swansea

Organizing Committee

José Espírito Santo	Centro de Matemática, University of Minho
Maria João Frade	HASLab, University of Minho and INESC TEC
Cláudia Mendes Araújo	Centro de Matemática, University of Minho
Luís Pinto	Centro de Matemática, University of Minho

Host

Centro de Matemática, University of Minho

Sponsors

COST Action CA15123 EUTypes

Câmara Municipal de Braga

Table of Contents

Invited talks

Contributed talks

Building verified cryptographic components using F*

Cédric Fournet

Microsoft Research, Cambridge, UK

The HTTPS ecosystem includes communications protocols such as TLS and QUIC, the X.509 public key infrastructure, and various supporting cryptographic algorithms and constructions. This ecosystem remains surprisingly brittle, with practical attacks and patches many times a year. To improve their security, we are developing high-performance, standards-compliant, formally verified implementation of these components. We aim for our verified components to be drop-in replacements suitable for use in mainstream web browsers, servers, and other popular tools. In this talk, I will give an overview of our approach and our results so far. I will present our verification toolchain, based on F*: a programming language with dependent types, programmable monadic effects, support for both SMT-based and interactive proofs, and extraction to C and assembly code. I will illustrate its application using security examples, ranging from the functional correctness of optimized implementations of cryptographic algorithms to the security of (fragments of) the new TLS 1.3 Internet Standard.

See https://fstar-lang.org/ for an online tutorial and research papers on F*, and https://project-everest.github.io/ for its security applications to cryptographic libraries, TLS, and QUIC.

I will discuss several ways of using machine learning to automate theorem proving and to help with automating formalization. The former includes learning to choose relevant facts for "hammer" systems, guiding the proof search of tableaux and superposition automated provers by learning from large ITP libraries, and guiding the application of tactics in interactive tactical systems. The latter includes learning probabilistic grammars from aligned informal/formal corpora and combining them with semantic pruning, and using recurrent neural networks to learn direct translation from Latex to formal mathematics. Finally, I will discuss systems that interleave learning and deduction in feedback loops, and mention some latest developments in these areas.

Multi Types for Higher-Order Languages

Delia Kesner

IRIF, CNRS and Universit Paris-Diderot, France

Quantitative techniques are emerging in different areas of computer science, such as model checking, logic, and automata theory, to face the challenge of today's resource aware computation.

In this talk we discuss multi types, a.k.a. non-idempotent (intersection) types, which provide a natural tool to reason about resource consumption. Multi types are applicable to a wide range of powerful models of computation, such as for example pattern matching, control operators and infinitary computations.

We provide a clean theoretical understanding of the use of resources, and survey some recent semantical and operational results in the domain.

The Predicative, Polymorphic Calculus of Cumulative Inductive Constructions and its implementation

Matthieu Sozeau

INRIA, France

In this presentation, I will give an overview of an extension of the Predicative Calculus of Inductive Constructions with polymorphic universes and cumulative inductive types, at the basis of Coq since version 8.7. Polymorphic universes with cumulativity allow the definition of constructions that are generic in their universes and constraints between them, while keeping at the source level the lightweight syntax of systems based on typical ambiguity. Cumulative inductive types extend this further to allow for very liberal (and somewhat surprising) type and term conversions that subsume the so-called template polymorphism feature of Coq.

We will focus in particular on the set-theoretic model of the calculus that justifies our treatment of cumulativity, and on the combined subtleties of universe polymorphism and Coq's higher-order unification algorithm. We will showcase the implementation through striking examples where universe polymorphism and cumulative inductive types are crucial: to express syntactical models of cumulative type theories and to faithfully follow informal mathematical practice while formalizing category theory.

This is joint work with Nicolas Tabareau, Amin Timany and Beta Ziliani.

Machine Learning for Proof Automation and Formalization

Josef Urban

Czech Institute of of Informatics, Robotics and Cybernetics (CIIRC), Czech Republic

I will discuss several ways of using machine learning to automate theorem proving and to help with automating formalization. The former includes learning to choose relevant facts for "hammer" systems, guiding the proof search of tableaux and superposition automated provers by learning from large ITP libraries, and guiding the application of tactics in interactive tactical systems. The latter includes learning probabilistic grammars from aligned informal/formal corpora and combining them with semantic pruning, and using recurrent neural networks to learn direct translation from Latex to formal mathematics. Finally, I will discuss systems that interleave learning and deduction in feedback loops, and mention some latest developments in these areas.

Martin Hofmann's case for non-strictly positive data types

Ralph Matthes

Institut de Recherche en Informatique de Toulouse (IRIT), CNRS and Université Paul Sabatier

In type theory, we normally want all structurally recursive programs to be terminating. To be a bit more precise, a walk through an inductive structure should stop after some time, and this independently from the specific code of the program or even the employed data structure. Least fixed points of antitone operations on types are therefore banned from type-theoretic systems. The considered operators should be monotone, and even syntactically so. In practice, this means that the expression that describes the operation should have its formal parameter only at positive positions. Positivity does not exclude going twice to the left of the arrow for the function type—only strict positivity would forbid that. The non-strictly positive data types may not have a naive set-theoretic semantics (as put forward by John Reynolds), but they exist well in system F (Jean-Yves Girard) in the sense that system F allows to encode them to form weakly initial algebras, in other words, as data types with constructors and an iterator for programming structurally recursive functions. As evaluation in system F is strongly normalizing, all those structurally recursive programs are terminating.

Why are programs with non-strictly positive data types so rare?

In his February 1993 note to the `TYPES forum` mailing list[1], Martin Hofmann crafted a program with a non-strictly positive data type out of an earlier one in ML (by Mads Tofte) that used a negative datatype. He claimed that he had "found a (sort of) reasonable program making nontrivial use of the datatype" he called `cont` (for continuations), which, in categorical data type notation would be

$$\mu X.1 + ((X \to L) \to L),$$

with L the type of lists over some given type. X only occurs non-strictly positively in the type expression $1 + ((X \to L) \to L)$.

The program Martin Hofmann reworked for this data type computes the list of entries of a binary labelled tree in the breadth first order. In that note, he presents LEGO (by Randy Pollack) code for its implementation in system F.

In a LaTeX draft of 5 pages "Approaches to Recursive Datatypes — a Case Study" (April 1995), Martin Hofmann shows how he got from the negative data type through dependent types to his proposal. There is also a sketch of a correctness proof by induction over binary trees (giving the intermediary lemmas to show).

While Martin Hofmann was still at Edinburgh University, these ideas created a lot of interest in the Mathematical Logic group at Munich University. Ulrich Berger found a verification of the algorithm that was based on a non-strictly positive inductive relation, while Anton Setzer clarified why the simple proof by Martin Hofmann worked: by identifying simpler "subtypes" that still served as invariants for the program. These developments took place still in 1995. In 2000-2002, I further simplified the prerequisites of the verification (the original one by Martin Hofmann assumed parametric equality), and I met Olivier Danvy in 2002 who saw that `cont` is a misnomer, and that one should consider it rather as a type of coroutines and transform other programs with coroutines into this style.

[1] http://www.seas.upenn.edu/~sweirich/types/archive/1993/msg00027.html

Instead, I advocated another non-strictly positive inductive type (previously considered by Christine Paulin-Mohring, as I understood later), in order to embed Michel Parigot's $\lambda\mu$-calculus into system F (published at TLCA 2001 and subsequent work in the post-proceedings of Logic Colloquium 2003[2]).

Although in 2001, Martin had become the holder of the Gentzen Chair at the University of Munich, and as such the leader of the group where I worked, I missed all opportunities to turn his inspiring notes into a paper together with him. Ironically, his ideas had influenced me more while he had been still in distant places. And this talk should remind the audience how much Martin's scientific insights were able to fascinate other researchers, even if they were not considered as ready to be published by Martin. Sadly, we have to live with these memories without further opportunities to get new notes from Martin or to work with him. May he rest in peace.

In a nutshell, I'll describe the breadth-first traversal algorithm by Martin Hofmann, how it can be verified, what is needed to do a verification in an intensional setting (system F without parametric equality) and what else could be programmed in this spirit. Time permitting, I'll allow myself some remarks on Martin Hofmann, as I have perceived him (as assistant in his research group).

[2] https://www.aslonline.org/books-lnl_24.html

Resourceful Dependent Types

Andreas Abel

Department of Computer Science and Engineering, Gothenburg University

Quantitative typing integrates resource usage tracking into the type system. The prime example is linear lambda-calculus that requires that each variable is used exactly once. Following Girard's initial proposal of linear logic, an abundance of substructural type systems have been proposed that allow fine control over resources. These type systems find their applications in many areas:

1. Compilation: Cardinality analyses such as strictness, absence (dead code), linearity, or affine usage [Verstoep and Hage, 2015] enable diverse optimizations in the generated code.
2. Security is an instance of absence analysis: Classified information may not flow to unclassified output.
3. Differential privacy [Reed and Pierce, 2010]: Fuzziness of values is captured by a metric type system, and sufficient fuzz guarantees privacy in information aggregation.

These type systems annotate variables in the typing context with usage information, i. e., the context is a finite map from variables x to their type A together with a resource value q. In the judgment $\Gamma \vdash t : C$, term t of type C can refer to variables x in Γ according to their usage description q.

The question how to extend quantitative typing to dependent types had been lacking a satisfactory answer for a long time. The problem can be traced back to the invariant of type theory that if $\Gamma \vdash t : C$, then $\Gamma \vdash C : \mathsf{Type}$, i. e., a well-typed term requires a well-formed type, *in the same context*. A judgement like $x :_1 A \vdash \mathsf{refl}\, x : (x \equiv_A x)$ violates this property, since x is used twice on the type side.

McBride [2016] cut the Gordian knot by decreeing that types are mental objects that do not consume resources. In particular, any reference to a variable in a type should count as 0. The invariant now only stipulates $0\Gamma \vdash_0 C : \mathsf{Type}$, which erases usage information by multiplication with 0, and can be seen as the vanilla, resource free typing judgement. McBride's approach has been refined by Atkey [2018] into two judgements $\vdash_1$ (resourceful) $\vdash_0$ (resourceless) where the latter is used for type constructors.

The radical solution however also has some drawbacks. By discounting resources in types in general, (1) we cannot utilize usage information for optimizing the computation of types during type-checking; and (2) we cannot interpret types as values, in particular, we lose the option to case on types. However, *type case* is useful at least in typed intermediate languages to derive specialized implementations of data structures and their operations for particular types.

We propose a fresh look at the problem of dependent resource typing, by decoupling resource information from the typing context Γ. Instead, we track usage information in terms and types separately by *resource contexts* γ and δ which map the variables x of Γ to resource quantities q. The invariant becomes $\Gamma \vdash t^{\gamma} : A^{\delta}$ implies $\Gamma \vdash A^{\delta} : \mathsf{Type}^{\emptyset}$, and no extra trickery is needed. Dependent function types $\Pi^{q,r}\, A\, F$ are indexed by two quantities: q describes how the function uses its argument of type A to produce the result; r describes how the codomain F uses that argument to produce the result type.

Resources q, r are drawn from a partially ordered commutative semiring which is positive ($q + r = 0$ implies $q = r = 0$) and free of zero dividers ($qr = 0$ implies $q = 0$ or $r = 0$). Ordering $q \leq r$ expresses that q is more precise as r, in the same sense as subtyping $A \leq B$ states that A

is more specific than B. A resource semiring to track linearity would be $\{\omega, 0, 1\}$ with $\omega \leq q$. It is a subsemiring of $\mathcal{P}(\mathbb{N})$ [Abel, 2015] with $\omega = \mathbb{N}$, $0 = \{0\}$, $1 = \{1\}$, pointwise addition and multiplication, and $\leq\ = \supseteq$.

In the following, we list the inference rules for judgment $\Gamma \vdash t^{\gamma} : A^{\delta}$ of a dependently typed lambda-calculus with a predicative universe hierarchy U_{ℓ}. Herein, we write $A \overset{q}{\to} B$ for the non-dependent function space $\Pi^{q,0} A\, (\lambda^{0} \_.\, B)$.

$$\text{UNIV}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{U}_{\ell}^{\emptyset} : \mathsf{U}_{\ell'}^{\emptyset}}\ \ell{<}\ell' \qquad \text{PI}\ \frac{\Gamma \vdash A^{\delta_1} : \mathsf{U}_{\ell}^{\emptyset} \qquad \Gamma \vdash F^{\delta_2} : A \overset{r}{\to} \mathsf{U}_{\ell}^{\delta_1}}{\Gamma \vdash (\Pi^{q,r} A\, F)^{\delta_1+\delta_2} : \mathsf{U}_{\ell}^{\emptyset}}$$

$$\text{VAR}\ \frac{\vdash \Gamma}{\Gamma \vdash x^{x:1} : A^{\delta}}\ x{:}A^{\delta} \in \Gamma \qquad \text{ABS}\ \frac{\Gamma,\, x{:}A^{\delta_1} \vdash t^{\gamma, x:q} : (F \cdot^{r} x)^{\delta_2, x:r}}{\Gamma \vdash (\lambda^{q} x.\, t)^{\gamma} : (\Pi^{q,r} A\, F)^{\delta_1+\delta_2}}$$

$$\text{APP}\ \frac{\Gamma \vdash t^{\gamma_1} : (\Pi^{q,r} A\, F)^{\delta_1+\delta_2} \qquad \Gamma \vdash u^{\gamma_2} : A^{\delta_1}}{\Gamma \vdash (t \cdot^{q} u)^{\gamma_1 + q\gamma_2} : (F \cdot^{r} u)^{\delta_2 + r\gamma_2}} \qquad \text{SUB}\ \frac{\Gamma \vdash t^{\gamma} : A^{\delta} \qquad \Gamma \vdash A^{\delta} \leq B^{\delta}}{\Gamma \vdash t^{\gamma} : B^{\delta}}$$

Subtyping $\Gamma \vdash A^{\delta} \leq A'^{\delta}$ takes imprecision in the resource information into account and is contravariant for function domains, as usual.

$$\frac{\Gamma \vdash A^{\delta} = A'^{\delta} : \mathsf{U}_{\ell}^{\emptyset}}{\Gamma \vdash A^{\delta} \leq A'^{\delta}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{U}_{\ell}^{\emptyset} \leq \mathsf{U}_{\ell'}^{\emptyset}}\ \ell \leq \ell'$$

$$\frac{\Gamma \vdash A'^{\delta_1} \leq A^{\delta_1} \qquad \Gamma,\, x{:}A'^{\delta_1} \vdash (F \cdot^{r} x)^{\delta_2, x:r} \leq (F' \cdot^{r} x)^{\delta_2, x:r}}{\Gamma \vdash (\Pi^{q,r} A\, F)^{\delta_1+\delta_2} \leq (\Pi^{q',r} A'\, F')^{\delta_1+\delta_2}}\ q' \leq q$$

Acknowledgments. This work was supported by Vetenskapsrådet under Grant No. 621-2014-4864 *Termination Certificates for Dependently-Typed Programs and Proofs via Refinement Types* and by the EU Cost Action CA15123 *Types for programming and verification.*

References

A. Abel. The next 700 modal type assignment systems. In T. Uustalu, editor, *21st Int. Conf. on Types for Proofs and Programs, TYPES 2015, Abstracts*. Institute of Cybernetics at Tallinn University of Technology, 2015. URL http://cs.ioc.ee/types15/abstracts-book/.

R. Atkey. The syntax and semantics of quantitative type theory. In *33rd ACM/IEEE Symp. on Logic in Computer Science (LICS'18)*, 2018. URL https://bentnib.org/quantitative-type-theory.html. To appear.

C. McBride. I got plenty o' nuttin'. In S. Lindley, C. McBride, P. W. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lect. Notes in Comput. Sci.*, pages 207–233. Springer, 2016. URL http://dx.doi.org/10.1007/978-3-319-30936-1_12.

J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In P. Hudak and S. Weirich, editors, *Proc. of the 15th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2010*, pages 157–168. ACM Press, 2010. URL http://doi.acm.org/10.1145/1863543.1863568.

H. Verstoep and J. Hage. Polyvariant cardinality analysis for non-strict higher-order functional languages: Brief announcement. In K. Asai and K. Sagonas, editors, *Proc. of the 2015 Wksh. on Partial Evaluation and Program Manipulation, PEPM, 2015*, pages 139–142. ACM Press, 2015. URL http://doi.acm.org/10.1145/2678015.2682536.

Sharing Equality is Linear*

Beniamino Accattoli[1], Andrea Condoluci[2], and Claudio Sacerdoti Coen[2]

[1] LIX – Inria & École Polytechnique – France
[2] Department of Computer Science and Engineering – University of Bologna – Italy

Reasonable evaluation and sharing. Somewhat counterintuitively, the λ-calculus is not a good setting for evaluating and representing higher-order programs—it is an excellent specification framework, but no practical tool implements it as it is. If it were a *reasonable* execution model, mechanising the evaluation sequence of a term t on random access machines (RAM) would have a cost polynomial in the size of t and in the number n of β-steps. In this way a program evaluating in a polynomial number of steps could indeed be considered as having polynomial cost, and the number of β-steps would become a reasonable complexity measure.

Unfortunately there is a problem called **size explosion**: there are families of terms whose size grows exponentially with the number of evaluation steps, obtained by nesting duplications one inside the other. In many cases **sharing** is the cure because size explosion is based on unnecessary duplications of subterms, that can be avoided if such subterms are instead shared, and evaluation is modified accordingly. The key point is that sharing-based implementations compute compact results, whose size does not explode. Sharing does indeed provide reasonable implementations for the λ-calculus: the first result for weak strategies is by Blelloch and Greiner in 1995 [4], and the first for strong strategies is by Accattoli and Dal Lago in 2014 [3]. In 2015, Accattoli and Sacerdoti Coen [1] showed that the optimisations for the strong case can be implemented with overhead *linear* in the size of the initial term and in the number of β-steps. A consequence of their result is that the size of the computed compact result $\texttt{nf}(t)$ is bilinear.

Reasonable conversion and sharing Some higher-order settings need more than evaluation of a single term: they also have to check whether two terms t and s are convertible—for instance to implement the equality predicate, as in Ocaml, or for type checking with dependent types, as in Coq. These settings usually rely on a set of folklore and ad-hoc heuristics for conversion, that quickly solve many frequent special cases. In the general case, however, one first evaluates t and s to their normal forms, and then checks them for equality—actually, for α-equality because terms in the λ-calculus are identified up to α (renaming of bound variables).

Size explosion forces reasonable evaluations to produce shared results. Sharing equality is α-equality of the underlying *unshared* results, that, for conversion to be reasonable, has to be testable in time polynomial in the sizes of $\texttt{nf}(t)$ and $\texttt{nf}(s)$. Since unfolding the sharing may take exponential time, sharing equality has to be tested *without* unfolding, which is tricky.

The literature contains only two algorithms explicitly addressing sharing equality, but none of these matches the complexity of evaluation. First, a quadratic algorithm by Accattoli and Dal Lago [2]; second, a $O(n \log n)$ algorithm by Grabmayer and Rochel [6], obtained by a reduction to equivalence of DFAs and treating the more general case of λ-terms with `letrec`. Thus the bottleneck for conversion seems to be deciding sharing equality.

Here, we provide the first algorithm that is linear in the size of the shared terms, improving over the literature: the complexity of sharing equality matches the one of evaluation, providing a combined bilinear algorithm for conversion, that is the real motivation behind this work.

This work is submitted to LICS 2018: the full version is on the first author's webpage. This work has been partially funded by the ANR JCJC grant COCA HOLA (ANR-16-CE40-004-01).

Computing Sharing Equality If a λ-term can be seen as a syntax tree whose root is the topmost constructor and whose leaves are variables, then a λ-term with sharing is a **DAG**: sharing of subterms is the fact that nodes may have more than one parent. Intuitively, two DAGs represent the same unfolded λ-term if they have the same structural paths, just collapsed differently. A natural way of checking sharing equality is to test DAGs for *bisimilarity*: our sharing equality is based on what we call **sharing equivalences**, that are bisimulations plus additional requirements about names—for α-equivalence—and that they are equivalence relations.

A key problem is the presence of **binders**, *i.e.* abstractions, and the fact that equality on λ-terms is α-equivalence. Graphically, it is standard to see abstractions as getting a backward edge from the variable they bound; this creates **cycles**, and the issue is that bisimilarity with cycles may not be linear—Hopcroft and Karp's algorithm [7], the best one, is only pseudo-linear, that is, with an inverse Ackermann factor. It turns out that cycles induced by binders are only a graphical representation of *scopes* and can be removed, but their essence remains: while two free variables are bisimilar only if they coincide, two bound variables are bisimilar only when also their binders are bisimilar. Scopes are characterised by a structural property called **domination**—exploring the DAG from top to bottom one necessarily visits the binder before the bound variable—and this property is the key ingredient for a linear algorithm.

Our **two-level linear algorithm** for sharing equality is inspired by Paterson and Wegman's (shortened PW) linear algorithm for first-order unification [8]. We pushed further the modularity suggested—but not implemented—by Calvès & Fernández in [5]:

- **Blind sharing check**: a reformulation of PW without the management of meta-variables for unification. It is a *first-order* test on λ-terms with sharing, checking that the unfolded terms have the same skeleton, but ignoring variable names. It actually computes the *sup* of the sharings of t and s, that is the *smallest* sharing equivalence between the two DAGs.
- **Name check**: a straightforward algorithm executed after the previous one, testing α-equivalence by checking that bisimilar bound variables have bisimilar binders and that two different free variables are never shared. The algorithm is complete thanks to domination under mild but key assumptions on the context in which terms are tested.

References

[1] Beniamino Accattoli and Claudio Sacerdoti Coen. On the relative usefulness of fireballs. In *LICS 2015*, pages 141–155, 2015.

[2] Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction. In *RTA*, pages 22–37, 2012.

[3] Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In *CSL-LICS '14*, pages 8:1–8:10, 2014.

[4] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995.

[5] Christophe Calvès and Maribel Fernández. The first-order nominal link. LOPSTR'10, pages 234–248, 2011.

[6] Clemens Grabmayer and Jan Rochel. Maximal sharing in the lambda calculus with letrec. In *ICFP 2014*, pages 67–80, 2014.

[7] J. Hopcroft and R. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 0, Dept. of Computer Science, Cornell U, December 1971.

[8] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158 – 167, 1978.

A modular formalization of bicategories in type theory *

Benedikt Ahrens[1] and Marco Maggesi[2]

[1] University of Birmingham, United Kingdom
b.ahrens@cs.bham.ac.uk
[2] Università degli Studi di Firenze, Italy
marco.maggesi@unifi.it

Abstract

We report on our ongoing effort to implement a library of bicategories in type theory, specifically in the UniMath library of univalent mathematics. It is developed in the context of a larger project aimed at defining signatures for dependent type theories and their models.

1 Introduction

Our goal is to study the categorical semantics of dependent type systems, in univalent type theory. A crucial step in this project is to set up a formal implementation of the fundamental category-theoretic definitions. Due to the complexity of the objects under consideration, it is of fundamental importance that constructions and reasoning can be carried out in a highly modular way. In this work, we present our attempt at developing such a modular formalization.

2 Background

The UniMath library of univalent mathematics [5] currently contains quite a few results on 1-category theory, but the theory of higher categories, in particular, of bicategories, is not well-developed yet.

Specifically, one development of bicategories in UniMath has been contributed by Mitchell Riley[1]. Riley gives the definition of "univalent bicategories", and shows that an equivalence of univalent categories induces an identity between them—a first step towards showing that the bicategory of univalent categories is univalent. However, his definition of bicategories—in the style of enrichment in 1-categories—seems not amenable to modular reasoning on these complex gadgets. Indeed, defining the hom-objects to be categories mixes data and properties in a way that makes it difficult to use in a proof-relevant setting.

In the this work, we present an alternative formalization of bicategories that avoids this problem. Additionally, it allows for modular construction of bicategories of complex objects using a bicategorical version of displayed categories by Ahrens and Lumsdaine [1].

3 Bicategories

The prevalent definition of bicategories in the literature, in the style of weak enrichment over 1-categories, is very concise and easy to check for correctness.

*Work on this project was done during a research visit of Marco Maggesi to Birmingham funded by the EUTypes project. Maggesi is grateful to MIUR and INdAM for continuous financial support. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0363.

[1] https://github.com/UniMath/UniMath/pull/409

Here, we follow an alternative presentation [4] where the structure of 0-cells (objects), 1-cells (morphisms), and 2-cells is made explict. This approach, in contrast to the former, adheres intrinsically to a general established design principle in intensional mathematics of strictly separating data and properties.

4 Displayed bicategories and modularity

Ahrens and Lumsdaine's displayed categories [1] allow for a modular construction of complex 1-categories from simpler ones, in "layers". A prototypical example comes from the stratification of structures in algebra. For instance, the construction of the category of groups from the category of sets plus some extra structure can easily be factorized and expressed via the "displayed group structure" over the displayed monoid structure on the category of sets.

Following this pattern, we give the bicategorical variant of displayed categories, and arrange the displayed 1-categories into a displayed bicategory over the bicategory of categories. We then systematically use displayed bicategories to implement several bicategorical constructions. Some currently implemented examples include direct product, sigma structures, and functor and cofunctor categories.

These constructions were enough to build in UniMath, in a modular way, the bicategory CwF of categories with families [2], in the reformulation by Fiore [3, Appendix].

5 Conclusions

Our code is available from the UniMath repository[2]. It consists of about 4,000 lines of code.

We are planning to extend our "zoo" of constructions of displayed bicategories. In particular, we are going to study univalent bicategories and modular ways of showing that a given bicategory, constructed from a displayed one, is univalent. Our project on signatures for dependent type theories will provide a challenging test-bed for the usability of our library.

We are grateful to Peter LeFanu Lumsdaine and Vladimir Voevodsky for helpful discussions on this subject.

References

[1] Benedikt Ahrens and Peter LeFanu Lumsdaine. "Displayed Categories". In: *Formal Structures for Computation and Deduction 2017*. Ed. by Dale Miller. Vol. 84. Leibniz International Proceedings in Informatics. Dagstuhl, Germany: Leibniz-Zentrum für Informatik, 2017, 5:1–5:16. DOI: 10.4230/LIPIcs.FSCD.2017.5.

[2] Peter Dybjer. "Internal Type Theory". In: *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*. 1995, pp. 120–134. DOI: 10.1007/3-540-61780-9_66.

[3] Marcelo Fiore. *Discrete Generalised Polynomial Functors*. Slides from talk at ICALP 2012, http://www.cl.cam.ac.uk/~mpf23/talks/ICALP2012.pdf.

[4] The *n*Lab. *Bicategory*.

[5] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath — a computer-checked library of univalent mathematics*. URL: https://github.com/UniMath/UniMath.

[2] https://github.com/UniMath/UniMath/pull/925

Specifying Quotient Inductive-Inductive Types

Thorsten Altenkirch[1], Paolo Capriotti[1], Gabe Dijkstra, Nicolai Kraus[1], and Fredrik Nordvall Forsberg[2]

[1] University of Nottingham
[2] University of Strathclyde

Abstract

We give a general specification of *quotient inductive-inductive types (QIITs)*. These are types and type families that are defined inductive-inductively, and simultaneously with their equalities. By construction, QIITs satisfy UIP. This talk is based on our paper [ACD+18].

The central concept of inductive definitions allows the construction of datatypes such as the natural numbers, lists and trees by presenting their constructors. We are interested in two generalisations: firstly, one can allow the mutual inductive definition of families of types, which may depend on each other; this is known as *inductive-inductive* definitions [NF13]. Secondly, the inductive definition of a type can include constructors for its equalities; such types are known as *higher inductive types* [LS17] in homotopy type theory. We combine both ideas and give a specification of inductive-inductive families with higher constructors, where it is built into the specification that the constructed types satisfy Uniqueness of Identity Proofs (UIP); we call such types *quotient inductive-inductive types (QIITs)*. An independent specification of QIITs via codes has recently been given by Kaposi and Kovács [KK18].

Despite not being backed by a specification (until now), the combination of induction-induction and higher constructors has already been explored. Examples include the Cauchy reals [Gil17] and the partiality monad [ADK17] (also doable without induction-induction [CUV17]). In both cases, QIITs allows avoiding the use of countable choice. A further example is the well-typed syntax of type theory in type theory [AK16], where a type of contexts and type families of types, terms and substitutions are defined simultaneously with their equalities.

We now give an overview of our framework for specifying QIITs; details and examples can be found in our paper [ACD+18]. A QIIT is specified by its *sort*, which encodes the types and type families that it consists of, and by a sequence of *constructors*

$$c : (x : F(X)) \to G(X, x) \tag{1}$$

that in turn are specified by appropriate *argument* and *target* functors F and G. Our main technical contribution is identifying a condition on the target functors G that lead to a well-behaved theory: we would like them to be continuous, i.e. preserve all limits. However, their domain might not have many limits to preserve, and so, we are led to the following refined notion (we write hSet for the category of types satisfying UIP):

Definition 1 (Relative continuity)**.** Let $\mathcal{C}$ be a category, $\mathcal{C}_0$ a complete category, and $U : \mathcal{C} \Rightarrow \mathcal{C}_0$ a functor. If I is a small category, and $X : I \Rightarrow \mathcal{C}$ is a diagram, we say that a cone $A \to X$ in $\mathcal{C}$ is a *U-limit cone*, or *limit cone relative to U*, if the induced cone $UA \to UX$ is a limit cone in $\mathcal{C}_0$. A functor $\mathcal{C} \Rightarrow \mathsf{hSet}$ is *continuous relative to U* if it maps U-limit cones to limit cones in hSet.

If $\mathcal{C}$ is complete and U creates limits, then relative continuity with respect to U reduces to ordinary continuity. Recall that the category of elements of a functor $F : \mathcal{C} \to \mathsf{hSet}$, denoted $\int^{\mathcal{C}} F$, has as objects pairs (X, x), where X is an object in $\mathcal{C}$, and $x : F\,X$. There is a forgetful functor $U : \int^{\mathcal{C}} F \Rightarrow \mathcal{C}$, and we will consider relative continuity with respect to this U, for complete categories $\mathcal{C}$. Using $\int^{\mathcal{C}} F$ as the domain of G in (1) means that the target can depend also on the argument $x : F(X)$.

Definition 2. A *constructor specification* (F, G) on a complete category $\mathcal{C}$ is given by:

- a functor $F : \mathcal{C} \Rightarrow \mathsf{hSet}$, called the *argument* functor of the specification;
- a relatively continuous functor $G : \int^{\mathcal{C}} F \Rightarrow \mathsf{hSet}$, called the *target* functor.

The intuition is that a constructor should only "construct" elements of one of the sorts, or equalities thereof; mathematically, relative continuity ensures that categories of QIIT algebras are complete (see below). As expected, point and path constructors (i.e. constructors targeting one of the sorts, or an equality in it, respectively) have relatively continuous target functors.

Lemma 3. *Target functors for point and path constructors are relatively continuous.* □

Hence our framework covers known examples of QIITs. Every constructor specification (F, G) gives rise to a category of algebras: objects are pairs (X, θ), where X is an object in $\mathcal{C}$, and $\theta : (x : F\,X) \to G(X, x)$ is a dependent function (in hSet).

Theorem 4. *The category of algebras for every constructor specification is complete.* □

This means that we can specify constructors on the category of algebras of the previously introduced constructors, meaning that later constructors can refer to previous ones. A QIIT is specified by a sequence of such constructors[1], with the intended semantics of the specification being the initial object in the ultimate category of algebras. The fact that the initial object is an algebra models the introduction rules of the QIIT, and initiality gives a "display map" formulation of the elimination rules:

Theorem 5. *A QIIT algebra X is initial if and only if it satisfies the* section induction principle*, stating that every algebra morphism into X has a section.* □

Unwinding the section induction principle, one can see the similarity with traditional formulations of elimination rules. We leave proving that initial algebras exist as future work; this will require further accessibility restrictions on the argument functors.

References

[ACD+18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In *FoSSaCS*, pages 293–310, 2018. Available as arXiv:1612.02346.

[ADK17] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited. In *FoSSaCS*, pages 534–549, 2017.

[AK16] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *POPL*, pages 18–29, 2016.

[CUV17] James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the delay monad by weak bisimilarity. *Mathematical Structures in Computer Science*, pages 1–26, 2017.

[Gil17] Gaëtan Gilbert. Formalising real numbers in homotopy type theory. In *CPP*, pages 112–124, 2017.

[KK18] Ambrus Kaposi and András Kovács. A syntax for higher inductive-inductive types. To appear at FSCD 2018. Available at `https://akaposi.github.io/hiit.pdf`, 2018.

[LS17] Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types, 2017. arXiv:1705.07088.

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

[1] A sequence of constructors is needed instead of a single constructor, because different constructors can have different targets, and later constructors can refer to previous constructors.

Constructing inductive-inductive types using a domain-specific type theory*

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

[1] University of Nottingham, United Kingdom
thorsten.altenkirch@nott.ac.uk
[2] Eötvös Loránd University, Budapest, Hungary
{divip|akaposi|kovacsandras}@inf.elte.hu

Inductive-inductive types (IITs, [4]) allow the mutual definition of a type and a family over the type, thus generalising mutual inductive types and indexed families of types. Examples of IITs are the well-typed syntax of type theory [2] or the dense completion of an ordered set [4, Example 3.3]. In this talk we *define* signatures for IITs using a *domain-specific type theory* (DSTT) where a context is a signature of an IIT. By induction on the syntax of the DSTT, we *describe* what it means to have constructors and dependent eliminator for each signature (following [3]). Then we show that the initial algebras and the dependent eliminator *exist*: the initial algebras will be given by terms of the DSTT and the eliminator will be given by the logical predicate interpretation [1]. Apart from having an embedded syntax of the DSTT, the metatheory is usual intensional type theory.

Signatures of inductive-inductive types. Our DSTT has an empty universe (we write underline for El) and a function space where the domain is small (that is, the domain can only be a neutral term) and there is no λ. We write $a \Rightarrow B$ when B does not mention the input. We write $\Gamma \vdash \sigma : \Delta$ for a context morphism from Γ to Δ (a list of terms which provide all types in Δ) and $A[\sigma]$, $t[\sigma]$ for substituted types and terms.

Contexts and variables: $\dfrac{}{\vdash \cdot}$ $\quad \dfrac{\Gamma \vdash A}{\vdash \Gamma, x : A}$ $\quad \dfrac{\Gamma \vdash A}{\Gamma, x : A \vdash x : A}$ $\quad \dfrac{\Gamma \vdash x : A \qquad \Gamma \vdash B}{\Gamma, y : B \vdash x : A}$

Universe: $\dfrac{}{\Gamma \vdash \mathsf{U}}$ $\quad \dfrac{\Gamma \vdash a : \mathsf{U}}{\Gamma \vdash \underline{a}}$

Functions with small domain: $\dfrac{\Gamma \vdash a : \mathsf{U} \qquad \Gamma, x : \underline{a} \vdash B}{\Gamma \vdash (x : a) \Rightarrow B}$ $\quad \dfrac{\Gamma \vdash t : (x : a) \Rightarrow B \qquad \Gamma \vdash u : \underline{a}}{\Gamma \vdash t\, u : B[x \mapsto u]}$

A context Γ in this DSTT is a signature for an IIT. For example, a subset of the intrinsic syntax of type theory (contexts and types) is given by the signature $\cdot$, $Con : \mathsf{U}$, $Ty : Con \Rightarrow \mathsf{U}$, $\bullet : \underline{Con}$, $\triangleright : (\Gamma : Con) \Rightarrow Ty\,\Gamma \Rightarrow \underline{Con}$, $U : (\Gamma : Con) \Rightarrow \underline{Ty\,\Gamma}$, $\Pi : (\Gamma : Con) \Rightarrow (A : Ty\,\Gamma) \Rightarrow Ty\,(\Gamma \triangleright A) \Rightarrow \underline{Ty\,\Gamma}$. Our simpler running example is $\Theta := (\cdot, N : \mathsf{U}, z : \underline{N}, s : N \Rightarrow \underline{N})$.

Notion of algebra ($-^{\mathsf{A}}$). By induction on the syntax of the DSTT we define the operation $-^{\mathsf{A}}$. Given a signature Γ (a context), Γ^{A} is the notion of algebra corresponding to it. The action of the operation on contexts, types, context morphisms and terms is specified as follows.

$$\frac{\vdash \Gamma}{\Gamma^{\mathsf{A}} \in \mathbf{Set}} \qquad \frac{\Gamma \vdash A}{A^{\mathsf{A}} \in \Gamma^{\mathsf{A}} \to \mathbf{Set}} \qquad \frac{\Gamma \vdash \sigma : \Delta}{\sigma^{\mathsf{A}} \in \Gamma^{\mathsf{A}} \to \Delta^{\mathsf{A}}} \qquad \frac{\Gamma \vdash t : A}{t^{\mathsf{A}} \in (\gamma \in \Gamma^{\mathsf{A}}) \to A^{\mathsf{A}}(\gamma)}$$

*This work was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

$-^{\mathsf{A}}$ is the standard (set-theoretic) interpretation of the DSTT. For example, $(\Gamma, x : A)^{\mathsf{A}} := (\gamma : \Gamma^{\mathsf{A}}) \times A^{\mathsf{A}}(\gamma)$, $((x : a) \to B)^{\mathsf{A}}(\gamma) := (\alpha \in a^{\mathsf{A}}(\gamma)) \to B^{\mathsf{A}}(\gamma, \alpha)$, $\mathsf{U}^{\mathsf{A}}(\gamma) := \mathbf{Set}$ and $\underline{a}^{\mathsf{A}}(\gamma) := a^{\mathsf{A}}(\gamma)$. Thus for our running example we get natural number algebras: $\Theta^{\mathsf{A}} = (n \in \mathbf{Set}) \times n \times (n \to n)$.

Notion of family over an algebra ($-^{\mathsf{F}}$). For a signature Γ, Γ^{F} is the logical predicate [1] over Γ^{A}. If $\gamma \in \Gamma^{\mathsf{A}}$ is the initial algebra, $\Gamma^{\mathsf{A}}(\gamma)$ gives the parameters of the eliminator (motives and methods). We have $\Theta^{\mathsf{F}}(n, z, s) = (n_F \in n \to \mathbf{Set}) \times n_F(z) \times \big((x : n) \to n_F(x) \to n_F(s(x))\big)$ for any $(n, z, s) \in \Theta^{\mathsf{A}}$. For a context Γ we have $\Gamma^{\mathsf{F}} \in \Gamma^{\mathsf{A}} \to \mathbf{Set}$ and for a context morphism $\Gamma \vdash \sigma : \Delta$ we get $\sigma^{\mathsf{F}} \in \Gamma^{\mathsf{F}}(\gamma) \to \Delta^{\mathsf{F}}(\sigma^{\mathsf{A}}(\gamma))$. We omit the specification for types and terms for reasons of space and we will do similarly for the operations below.

Notion of section ($-^{\mathsf{S}}$). For a signature Γ, an algebra $\gamma \in \Gamma^{\mathsf{A}}$ and a family $\gamma_F \in \Gamma^{\mathsf{F}}(\gamma)$, $\Gamma^{\mathsf{S}}(\gamma, \gamma_F)$ is the type of sections from γ to γ_F. For Θ this says that given an algebra (n, z, s) and a family (n_F, z_F, s_F) over it, we get a function n_F which maps z to z_F and s to s_F: $\Theta^{\mathsf{S}}((n, z, s), (n_F, z_F, s_F)) = \big(n_S \in (\alpha \in n) \to n_F(\alpha)\big) \times \big(n_S(z) = z_F\big) \times \big(n_S(s(x)) = s_F(n_S(x))\big)$.

Existence of constructors ($-^{\mathsf{C}-}$). For every signature Γ, we would like to have an element of Γ^{A}, the initial algebra or the type and term constructors of the IIT. We define these as terms of the DSTT. E.g. for Θ, we would like to generate $(\{t \,|\, \Theta \vdash t : \underline{n}\}, z, \lambda x.s\, x) \in \Theta^{\mathsf{A}}$. We fix a signature Ω and define the operator $-^{\mathsf{C}\Omega}$ by induction on the syntax of the DSTT. For a context Γ, we have $\Gamma^{\mathsf{C}\Omega} \in (\Omega \vdash \sigma : \Gamma) \to \Gamma^{\mathsf{A}}$ and for a type $\Gamma \vdash A$, we have $A^{\mathsf{C}\Omega} \in (\Omega \vdash \sigma : \Gamma) \times (\Gamma \vdash t : A[\sigma]) \to A^{\mathsf{A}}(\Gamma^{\mathsf{C}\Omega}(\sigma))$. On the universe, $-^{\mathsf{C}\Omega}$ returns terms of the given type: $\mathsf{U}^{\mathsf{C}\Omega}(\sigma, a) := \{t \,|\, \Omega \vdash t : \underline{a}\}$, and for $\underline{a}$ it simply returns the term: $\underline{a}^{\mathsf{C}\Omega}(\sigma, t) := t$. The initial Ω-algebra is given by $\Omega^{\mathsf{C}\Omega}(\mathsf{id}_\Omega) \in \Omega^{\mathsf{A}}$ where id_Ω is the identity context morphism on Ω.

Existence of dependent eliminator ($-^{\mathsf{E}-}$). After fixing a signature Ω and given motives and methods $\omega \in \Omega^{\mathsf{F}}(\Omega^{\mathsf{C}}(\mathsf{id}_\Omega))$, we define the operation $-^{\mathsf{E}\omega}$ by induction on the syntax of the DSTT. For a context Γ it gives $\Gamma^{\mathsf{E}\omega} \in (\Omega \vdash \sigma : \Gamma) \to \Gamma^{\mathsf{S}}(\Gamma^{\mathsf{C}}(\sigma), \sigma^{\mathsf{F}}(\omega))$. $\mathsf{U}^{\mathsf{E}\omega}$ works by calling the $-^{\mathsf{F}}$ operation on its input. Again, the identity substitution is used when deriving the eliminator itself (note that ${\mathsf{id}_\Omega}^{\mathsf{F}}(\omega) = \omega$): $\Omega^{\mathsf{E}\omega}(\mathsf{id}_\Omega) \in \Omega^{\mathsf{S}}(\Omega^{\mathsf{C}}(\mathsf{id}_\Omega), \omega)$. For our example, given $\theta \in \Theta^{\mathsf{F}}(\Theta^{\mathsf{C}}(\mathsf{id}_\Theta))$, we get $\Theta^{\mathsf{E}\theta}(\mathsf{id}_\Theta) = \big(\lambda t.\mathsf{transport}(t^{\mathsf{C}}(\mathsf{id}_\Theta), t^{\mathsf{F}}(\theta)), \mathsf{refl}, \lambda t\, t_F.\mathsf{J}(\mathsf{refl}, t^{\mathsf{C}}(\mathsf{id}_\Theta))\big)$.

Further work. An Agda formalisation of the operations $-^{\mathsf{C}-}$ and $-^{\mathsf{E}-}$ is on the way (the previous operations are formalised). We would also like to extend them to a larger DSTT describing infinitary constructors and equality constructors as well (quotient inductive-inductive types). A Haskell implementation of a type theory with IITs using this approach would be also interesting to experiment with.

References

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

[2] James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, January 2009.

[3] Ambrus Kaposi and András Kovács. A syntax for higher inductive-inductive types. FSCD 2018.

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

Reducing Inductive-Inductive Types to Indexed Inductive Types

Thorsten Altenkirch[1], Ambrus Kaposi[2], András Kovács[2], and Jakob von Raumer[1]

[1] University of Nottingham, United Kingdom
thorsten.altenkirch@nott.ac.uk, jakob@von-raumer.de
[2] Eötvös Loránd University, Budapest, Hungary
{akaposi, kovacsandras}@inf.elte.hu

1 Motivation

Many dependently typed languages which are built on foundations like the calculus of constructions (CoC) provide support for indexed inductive types. These are type families which are inductively defined using constructors that create an instance over arbitrary elements of the base type (the type of $\mathbb{N}$-indexed vectors being a prominent example). Most of these languages, e. g. Coq [2] and Lean [3], don't allow for so called inductive-inductive types in which, for example, the user mutually defines a type A and a family B in which A may appear in the index, and where constructors of A and B may refer to the constructors of each other.

Use cases for inductive-inductive types encompass important applications like the internalization of the syntax of dependent type theory itself ("type theory in type theory" [1]) and the definition of the Cauchy construction of the real numbers [7]. We thus ask the question whether inductive-inductive types can be emulated in a language that only provides indexed inductive types, that is, most importantly, how to construct an appropriate eliminator for these types. The quesion whether this is possible has been brought up in previous studies on inductive-inductive types [6, 4].

2 Approach

We are given a list of *sorts* which we want to define and a list of *constructors*, each with potential references to others. First we define the category of *typed algebras*, of which we aim to construct the initial element. Next, we also define *untyped algebras* which we obtain by erasing all the indices from sorts and constructors. The initial object can be constructed using indexed induction only. To make up for the missing indexing in the untyped algebras, we introduce an inductively defined *well-typedness predicate* which contains the information that a given element of an untyped sort is really indexed by a given index element. The desired initial object of the category of typed algebras is then given by using this predicate to filter for well-typed elements. To prove initiality, we construct an inductive relation between elements of the initial untyped algebra and elements of an arbitrary typed algebra. We then show that this relation is functional.

3 Results & Future Work

The approach has been formalized for the example of a fragment of an inductive-inductive syntax of type theory in Agda, and ported to Lean. We aim to use Lean's meta-language [5] to

automate the construction and provide a user-defined command to create inductive-inductive types.

4 Example

As a first example, we looked at the type Con of *contexts* and the type Ty of *types* in a formalized syntax of a type theory, with a constructor $\mathsf{nil} : \mathsf{Con}$ for an empty context, $\mathsf{ext} : \prod_{\Gamma:\mathsf{Con}} \mathsf{Ty}(\Gamma) \to \mathsf{Con}$ for context extension, $\mathsf{unit} : \prod_{\Gamma:\mathsf{Con}} \mathsf{Ty}(\Gamma)$ for an atomic unit type and $\mathsf{pi} : \prod_{\Gamma:\mathsf{Con},A:\mathsf{Ty}(\Gamma)} \mathsf{Ty}(\mathsf{ext}(\Gamma, A)) \to \mathsf{Ty}(\Gamma)$ for a Π-type. For this example the typed and untyped algebras are represented by the following Lean code:

```
structure CT :=
  (C : Type u)
  (T : C → Type u)
  (nil : C)
  (ext : Π Γ, T Γ → C)
  (unit : Π (Γ : C), T Γ)
  (pi : Π Γ A,
    T (ext Γ A) → T Γ)

structure CT′ :=
  (C : Type u)
  (T : Type u)
  (nil : C)
  (ext : C → T → C)
  (unit : C → T)
  (pi : C → T → T → T)
```

```
inductive S′0 : bool → Type u
| nil : S′0 ff
| ext : S′0 ff → S′0 tt → S′0 ff
| unit : S′0 ff → S′0 tt
| pi : S′0 ff → S′0 tt → S′0 tt → S′0 tt

parameters (M : CT)
def rel_arg : bool → Type u
| ff := M.C
| tt := Σ γ, M.T γ
inductive rel : Π b, S′0 b → rel_arg b → Prop
| nil : rel ff S′0.nil M.nil
| ext : Π Γ A γ a, rel ff Γ γ → rel tt A ⟨γ, a⟩
    → rel ff (S′0.ext Γ A) (M.ext γ a)
| unit : Π Γ γ, rel ff Γ γ → rel tt (S′0.unit Γ) ⟨γ, M.unit γ⟩
| pi : Π Γ A B γ a b, rel ff Γ γ → rel tt A ⟨γ, a⟩ →
    rel tt B ⟨M.ext γ a, b⟩ → rel tt (S′0.pi Γ A B) ⟨γ, M.pi γ a b⟩
```

References

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

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

[3] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015.

[4] Gabe Dijkstra. *Quotient inductive-inductive definitions*. PhD thesis, University of Nottingham, 2017.

[5] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):34, 2017.

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

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

Semantic subtyping for non-strict languages

Davide Ancona[1], Giuseppe Castagna[2], Tommaso Petrucciani[1,2], and Elena Zucca[1]

[1] DIBRIS, Università di Genova, Italy
[2] IRIF, CNRS and Université Paris Diderot, France

Semantic subtyping is an approach to define type systems featuring union and intersection types and a precise subtyping relation. It has been developed for strict languages, and it is unsound for non-strict semantics. We describe how to adapt it to languages with lazy evaluation.

Semantic subtyping. Union and intersection types can be used to type several language constructs – from branching and pattern matching to overloading – very precisely. However, they make it challenging to define a subtyping relation that behaves precisely and intuitively.

Semantic subtyping is a technique to do so, studied by Frisch et al. [1] for types given by:

$$t ::= b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathsf{Empty} \mid \mathsf{Any} \qquad \text{where } b ::= \mathsf{Int} \mid \mathsf{Bool} \mid \cdots$$

Types include constructors – basic types b, arrows, and products – plus union $t \vee t$, intersection $t \wedge t$, negation (or complementation) $\neg t$, and the bottom and top types Empty and Any (actually, $t_1 \wedge t_2$ and Any can be defined as $\neg(\neg t_1 \vee \neg t_2)$ and $\neg\mathsf{Empty}$). Types can also be recursive (simply, by considering the types that are *coinductively* generated by the productions above).

Subtyping is defined by giving an interpretation $[\![\cdot]\!]$ of types as sets and defining $t_1 \leq t_2$ as $[\![t_1]\!] \subseteq [\![t_2]\!]$. Intuitively, we can see $[\![t]\!]$ as the set of values which inhabit t in the language. By interpreting union, intersection, and negation as the corresponding operations on sets, we ensure that subtyping will satisfy all commutative and distributive laws we expect (e.g., $(t_1 \times t_2) \vee (t'_1 \times t'_2) \leq (t_1 \vee t'_1) \times (t_2 \vee t'_2)$ or $(t \to t_1) \wedge (t \to t_2) \leq t \to (t_1 \wedge t_2)$).

This relation is used to type a call-by-value language featuring higher-order functions, data constructors and destructors (pairs), and a typecase construct which models runtime type dispatch and acts as a form of pattern matching. Functions can be recursive and are explicitly typed: their type can be an intersection of arrow types, describing overloaded behaviour.

Semantic subtyping in lazy languages. Current semantic subtyping systems are unsound for non-strict semantics because of the way they deal with the bottom type Empty, which corresponds to the empty set of values ($[\![\mathsf{Empty}]\!] = \varnothing$). The intuition is that a (reducible) expression e can be safely given a type t only if, whenever e returns a result, this result is a value in t. Accordingly, Empty can only be assigned to expressions that are statically known to diverge (i.e., that never return a result). For example, the ML expression `let rec f x = f x in f ()` has type Empty. Let $\bar{e}$ be this expression and consider the following typing derivations, which are valid in semantic subtyping systems (π_2 projects the second component of a pair).

$$\dfrac{\dfrac{\vdash (\bar{e}, 3) : \mathsf{Empty} \times \mathsf{Int}}{\vdash (\bar{e}, 3) : \mathsf{Empty} \times \mathsf{Bool}}\,[\simeq]}{\vdash \pi_2\,(\bar{e}, 3) : \mathsf{Bool}} \qquad \dfrac{\dfrac{\vdash \lambda x.\,3 : \mathsf{Empty} \to \mathsf{Int}}{\vdash \lambda x.\,3 : \mathsf{Empty} \to \mathsf{Bool}}\,[\simeq] \qquad \vdash \bar{e} : \mathsf{Empty}}{\vdash (\lambda x.\,3)\,\bar{e} : \mathsf{Bool}}$$

Note that both $\pi_2\,(\bar{e}, 3)$ and $(\lambda x.\,3)\,\bar{e}$ diverge in call-by-value semantics (since $\bar{e}$ must be evaluated first), while they both reduce to 3 in call-by-name or call-by-need. The derivations are therefore sound for call-by-value, while they are clearly unsound with non-strict evaluation.

Why are these derivations valid? The crucial steps are those marked with $[\simeq]$, which convert between types that have the same interpretation; $\simeq$ denotes this equivalence relation. With

semantic subtyping, $\mathsf{Empty} \times \mathsf{Int} \simeq \mathsf{Empty} \times \mathsf{Bool}$ holds because all types of the form $\mathsf{Empty} \times t$ are equivalent to Empty itself: none of these types contains any value (indeed, product types are interpreted as Cartesian products). It can appear more surprising that $\mathsf{Empty} \to \mathsf{Int} \simeq \mathsf{Empty} \to \mathsf{Bool}$ holds. We interpret a type $t_1 \to t_2$ as the set of functions which, on arguments of type t_1, return results in type t_2. Since there is no argument of type Empty (because, in call-by-value, arguments are always values), all types of the form $\mathsf{Empty} \to t$ are equivalent.

In passing, note that somewhat similar problems occur when using refinement types for non-strict semantics, as studied by Vazou et al. [2] (the false refinement is analogous to Empty).

Our approach. The intuition behind our solution is that, with non-strict semantics, it is not appropriate to see a type as the set of the values that have that type. In a call-by-value language, operations like application or projection must occur on values: thus, we can identify two types if they contain the same values. In non-strict languages, instead, operations also occur on partially evaluated results which might contain diverging sub-expressions.

The Empty type is important for the internal machinery of subtyping. Notably, the decision procedure for subtyping relies on the existence of types with empty interpretation (e.g., $t_1 \leq t_2$ if and only if $t_1 \wedge \neg t_2$ is empty). In a strict setting, it is sound to assign Empty to diverging computations. In a non-strict one, though, Empty should be completely empty: no expression at all should inhabit it. Diverging expressions should have a different type, with non-empty interpretation. We add this new type, written $\bot$, and have it be non-empty but disjoint from the types of constants, functions, and pairs: $[\![\bot]\!]$ is a singleton whose element represents divergence.

Introducing the $\bot$ type means that we track termination in types, albeit very approximately. We allow the derivation of types like Int, $\mathsf{Int} \to \mathsf{Bool}$, or $\mathsf{Int} \times \mathsf{Bool}$ (which are disjoint from $\bot$) only for expressions that are statically guaranteed to terminate. In particular, our rules can only derive them for constants, functions, and pairs. Application and projection, instead, always have types of the form $t \vee \bot$, meaning that they could diverge. We allow the typing rules to propagate the $\bot$ type. For example, to type the application $e_1\, e_2$, if $\Gamma \vdash e_2 \colon t$, we require $\Gamma \vdash e_1 \colon (t \to t') \vee \bot$ instead of $\Gamma \vdash e_1 \colon t \to t'$, so that e_1 is allowed to be possibly diverging. Then, $e_1\, e_2$ has type $t' \vee \bot$. Subtyping verifies $t \leq t \vee \bot$, so a terminating expression can be used where a possibly diverging one is expected. Because $[\![\bot]\!]$ is non-empty, the problematic type equivalences we have seen do not hold for it. Indeed, $\bot \times \mathsf{Int}$ is now the type of pairs whose first component diverges and the second evaluates to an Int: it is not equivalent to $\bot \times \mathsf{Bool}$.

Typing all applications as possibly diverging is a very coarse approximation, but it achieves our goal of giving a sound type system that still enjoys most benefits of semantic subtyping. Indeed, $\bot$ can be seen as an "internal" type that does not need to be written explicitly by programmers. As future work we intend to explore whether we can give a more expressive system, while maintaining soundness, by tracking termination somewhat more precisely, and to give a semantic interpretation of types in terms of sets of "results" rather than of "values".

We have defined this type system for a call-by-need variant of the language studied by Frisch et al. [1], and we have proved its soundness. The choice of call-by-need stems from the presence of union and intersection types: indeed, for quite technical reasons, our system is not sound for call-by-name if we assume that reduction might be non-deterministic.

References

[1] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–67, 2008.

[2] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell. In *ICFP '14*, 2014.

The Later Modality in Fibrations

Henning Basold

CNRS, ENS Lyon, henning.basold@ens-lyon.fr

In this talk, we will study the later modality in fibrations. What would be the motivation for this? In a type-theoretic setting, the later modality was first introduced by Nakano [Nak00] and afterwards studied by Appel et al. [App+07], Atkey and McBride [AM13], Møgelberg [Møg14] and Birkedal et al. [Bir+16]. On the semantic side, Milius and Litak [ML17] have studied the later modality axiomatically, while Birkedal et al. [Bir+12; BM13] studied it for ω^{op}-chains in the category of sets, called the topos of trees. Since the category of sets provides a very rich setting for higher-order logic, one can encode most propositions and proofs in that category. However, syntactic presentations get lost through this encoding. This brings us to the motivation of this work: extending an arbitrary (syntactic) logic with a later modality, independently of the presentation of the logic.

As it turns out, fibrations provide us a good basis for this project, since they allow us to deal abstractly with formulas that contain typed variables. For instance, one can organise syntactic first-order logics, dependent type theories, quantitative predicates etc. into fibrations. More precisely, let $p\colon \mathbf{E} \to \mathbf{B}$ be functor. For $I \in \mathbf{B}$, we let $\mathbf{E}_I$ be the *fibre above* I that has objects $X \in \mathbf{E}$ with $p(X) = I$ and morphisms $f\colon X \to Y$ that fulfil $p(f) = \mathrm{id}_I$. A *(cloven) fibration* is a functor $p\colon \mathbf{E} \to \mathbf{B}$, such that for every morphism $u\colon I \to J$ in $\mathbf{B}$ there is a functor $u^*\colon \mathbf{E}_J \to \mathbf{E}_I$ with isomorphisms $\mathrm{id}_I^* \cong \mathrm{Id}_{\mathbf{E}_I}$ and $u \circ v^* \cong v^* \circ u^*$ that fulfil certain coherence conditions. The functors u^* are called *reindexing* or *substitution* functors. In particular, we are interested in fibrations that are fibred Cartesian closed categories (fibred CCCs), which intuitively means that we can form the conjunction and implication of formulas in the same context. More technically, a fibration is a *fibred CCC* if every fibre $\mathbf{E}_I$ has finite products and exponential objects that are preserved by reindexing functors. The setting of fibred CCCs will allow us to prove the usual rules of the later modality.

In our study of the later modality, we proceed as follows. Given a category $\mathbf{B}$, an ω^{op}-*chain* in $\mathbf{B}$ is a functor $c\colon \omega^{\mathrm{op}} \to \mathbf{B}$, and we denote by $\overline{\mathbf{B}}$ the functor category $[\omega^{\mathrm{op}}, \mathbf{B}]$ that has chains as objects and natural transformations as morphisms. We show how to obtain from a fibration $p\colon \mathbf{E} \to \mathbf{B}$ a fibration $\overline{p}\colon \overline{\mathbf{E}} \to \overline{\mathbf{B}}$ of ω^{op}-chains. In this fibration, we can define for each chain $c\colon \omega^{\mathrm{op}} \to \mathbf{B}$ the later modality as a fibred functor $\blacktriangleright\colon \overline{\mathbf{E}}_c \to \overline{\mathbf{E}}_c$ and its unit $\mathrm{next}\colon \mathrm{Id} \Rightarrow \blacktriangleright$. That this functor is fibred intuitively means that substitutions distribute over the later modality via $\mathrm{subst}_\sigma\colon u^*(\blacktriangleright \sigma) \cong \blacktriangleright(u^* \sigma)$. The functor $\blacktriangleright$ also preserves fibred (finite) products. To be able to express and solve fixed point equations we need exponential objects, because for an ω^{op}-chain σ in $\mathbf{E}$, the solutions of contractive fixed point equations on σ are given by a morphism $\mathrm{löb}_\sigma\colon \sigma^{\blacktriangleright \sigma} \to \sigma$, cf. [Bir+12; ML17]. Thus, in the next step we show that the fibration $\overline{p}$ is a fibred CCC and that the morphism $\mathrm{löb}_\sigma$ exists for each σ, that löb is dinatural in σ and that it can be used to solve contractive equations. These constructions are summed up in the rules below, where we leave the usual category theoretical constructions and the equations out. Note that the rules are proof-relevant, hence apply also to type-theoretic settings.

$$\frac{f\colon \tau \to \sigma}{\blacktriangleright f\colon \blacktriangleright \tau \to \blacktriangleright \sigma} \qquad \frac{f\colon \tau \to \blacktriangleright \sigma \times \blacktriangleright \delta}{\iota^{-1} \circ f\colon \tau \to \blacktriangleright(\sigma \times \delta)} \qquad \frac{f\colon \tau \to \blacktriangleright(u^* \sigma)}{\mathrm{subst}_\sigma^{-1} \circ f\colon \tau \to u^*(\blacktriangleright \sigma)}$$

$$\frac{f\colon \tau \to \sigma}{\mathrm{next}_\sigma \circ f\colon \tau \to \blacktriangleright \sigma} \qquad \frac{f\colon \tau \times \blacktriangleright \sigma \to \sigma}{\mathrm{löb}_\sigma \circ \lambda f\colon \tau \to \sigma}$$

Above, we described the later modality and solutions to fixed point equations in general. The reason for introducing all this machinery is to be able to construct morphisms into coinductive predicates. Let $X \in \mathbf{B}$, we denote by $\overline{\mathbf{E}}_X$ the fibre above the constant chain K_X. A *coinductive predicate* is a final coalgebra $\nu\Phi$ for a functor $\Phi\colon \overline{\mathbf{E}}_X \to \overline{\mathbf{E}}_X$. If $\nu\Phi$ can be constructed as limit of the ω^{op}-chain $\overleftarrow{\Phi}$, then morphisms $\psi \to \nu\Phi$ in $\mathbf{E}_X$ are equivalently given by morphisms $K_\psi \to \overleftarrow{\Phi}$ in $\overline{\mathbf{E}}_X$. Moreover, if we write $\overline{\Phi}$ for the pointwise application of Φ, then we have $\overleftarrow{\Phi} = \blacktriangleright\left(\overline{\Phi}\overleftarrow{\Phi}\right)$. Finally, given a functor $T\colon \overline{\mathbf{E}}_X \to \overline{\mathbf{E}}_X$, we say that T is Φ-*compatible* if there is a natural transformation $\rho\colon T\Phi \Rightarrow \Phi T$. For a compatible T, it is easy to construct a morphism $\overleftarrow{\rho}\colon \overline{T}\left(\overleftarrow{\Phi}\right) \to \overleftarrow{\Phi}$. Putting all of this together, we obtain the following rules.

$$\frac{K_\psi \to \overleftarrow{\Phi}}{\psi \to \nu\Phi} \qquad \frac{f\colon \tau \to \blacktriangleright\left(\overline{\Phi}\overleftarrow{\Phi}\right)}{f\colon \tau \to \overleftarrow{\Phi}} \qquad \frac{f\colon \tau \to \overline{T}\overleftarrow{\Phi} \qquad \rho\colon T\Phi \Rightarrow \Phi T \ (T \text{ compatible})}{\overleftarrow{\rho} \circ f\colon \tau \to \overleftarrow{\Phi}}$$

In the talk, I will explain these constructions, the lifting of quantifiers to $\overline{p}$, some illustrative examples and how to obtain a syntax if the fibration p arises from a syntactic logic. Further details and the extension from ω to any well-founded set are in the preprint [Bas18].

References

[AM13] Robert Atkey and Conor McBride. "Productive Coprogramming with Guarded Recursion". In: *ICFP*. ACM, 2013, pp. 197–208. DOI: 10.1145/2500365.2500597.

[App+07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards and Jérôme Vouillon. "A Very Modal Model of a Modern, Major, General Type System". In: *POPL*. ACM, 2007, pp. 109–122. DOI: 10.1145/1190216.1190235.

[Bas18] Henning Basold. "Breaking the Loop: Recursive Proofs for Coinductive Predicates in Fibrations". In: *ArXiv e-prints* (Feb. 2018). arXiv: 1802.07143 [cs.LO].

[Bir+12] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer and Kristian Støvring. "First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees". In: *LMCS* 8.4 (2012). DOI: 10.2168/LMCS-8(4:1)2012.

[Bir+16] Lars Birkedal, Ales Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters and Andrea Vezzosi. "Guarded Cubical Type Theory: Path Equality for Guarded Recursion". In: *CSL 2016*. Vol. 62. LIPIcs. Schloss Dagstuhl, 2016, 23:1–23:17. ISBN: 978-3-95977-022-4. DOI: 10.4230/LIPIcs.CSL.2016.23.

[BM13] Lars Birkedal and Rasmus Ejlers Møgelberg. "Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes". In: *LICS*. IEEE Computer Society, 2013, pp. 213–222. DOI: 10.1109/LICS.2013.27.

[ML17] Stefan Milius and Tadeusz Litak. "Guard Your Daggers and Traces: Properties of Guarded (Co-)Recursion". In: *Fundam. Inform.* 150.3-4 (2017), pp. 407–449. DOI: 10.3233/FI-2017-1475. arXiv: 1603.05214.

[Møg14] Rasmus Ejlers Møgelberg. "A Type Theory for Productive Coprogramming via Guarded Recursion". In: *CSL-LICS*. ACM, 2014, 71:1–71:10. DOI: 10.1145/2603088.2603132.

[Nak00] Hiroshi Nakano. "A Modality for Recursion". In: *LICS*. IEEE Computer Society, 2000, pp. 255–266. DOI: 10.1109/LICS.2000.855774.

Dependent Right Adjoint Types

Lars Birkedal[1], Ranald Clouston[1], Bassel Mannaa[2], Rasmus Ejlers Møgelberg[2], Andrew M Pitts[3], and Bas Spitters[1]

[1] Aarhus University, Aarhus Denmark
{birkedal,ranald.clouston,spitters}@cs.au.dk
[2] IT University of Copenhagen Copenhagen, Denmark
{basm,mogel}@itu.dk
[3] Cambridge University, Cambridge, U.K.
andrew.pitts@cl.cam.ac.uk

There is recent interest in modal type theories, e.g. guarded type theory, nominal type theory [6], clocked type theory [5], cohesive type theory and cubical type theory. In this work we describe the model theory of Fitch-style [2] modal dependent type theories. These include nominal type theories and (more recently) clocked type theory. The goal of this work is to isolate and study a common construction in the models of these modal type theories. The construction centers around an adjoint pair of operators, where the left adjoint is an operation on contexts and the right adjoint is an operation on families. For example in nominal type theory these are given by the rules

$$\frac{\Gamma \vdash \qquad n \notin \Gamma}{\Gamma, [n : \mathsf{N}]} \qquad \frac{\Gamma, [n : \mathsf{N}] \vdash \mathsf{A}}{\Gamma \vdash \mathit{И}[n : \mathsf{N}]\mathsf{A}}$$

One can then think of the model construction of these calculi as one of finding a dependent form of the adjunction

$$\frac{LA \to B}{A \to RB}$$

That is, when the function space is given by a Π-type.

We formulate the theory of these dependent adjunctions in the (aptly named) Categories with families with dependent right adjoints (CwDRA).

Definition 1 (CwDRA)**.** A CwDRA is a CwF [3] with the following added structure:
A functor L:

$$\frac{\Gamma \vdash}{\mathsf{L}\Gamma \vdash} \qquad \frac{\gamma : \Delta \to \Gamma}{\mathsf{L}\gamma : \mathsf{L}\Delta \to \mathsf{L}\Gamma} \qquad \mathsf{Lid} = \mathsf{id} \qquad \mathsf{L}(\gamma \circ \delta) = \mathsf{L}\gamma \circ \mathsf{L}\delta$$

An operation on families:

$$\frac{\mathsf{L}\Gamma \vdash A}{\Gamma \vdash \mathsf{R}_\Gamma A} \qquad (\mathsf{R}_\Gamma A)[\gamma] = \mathsf{R}_\Delta(A[\mathsf{L}\gamma])$$

An invertible transpose operation on elements of families:

$$\frac{\mathsf{L}\Gamma \vdash a : A}{\Gamma \vdash \overline{a} : \mathsf{R}_\Gamma A} \qquad \frac{\Gamma \vdash b : \mathsf{R}_\Gamma A}{\mathsf{L}\Gamma \vdash \overline{b} : A} \qquad \overline{a[\mathsf{L}\gamma]} = \overline{a}[\gamma] \qquad \overline{\overline{a}} = a$$

Lemma 1. *If* $\Gamma \vdash b : \mathsf{R}_\Gamma A$ *and* $\gamma : \Delta \to \Gamma$*, then* $\overline{b[\gamma]} = \overline{b}[\mathsf{L}\gamma]$

Although not provided here, we also describe a syntax and term model of CwFDRA which can be seen as a dependent version of the ones in [2].

In the model construction of these type theories it is often the case that one starts with adjoint pair of endofunctors on the category of contexts. It is then natural to ask, what are the sufficient (resp. necessary) conditions under which this adjunction lifts to a dependent adjunction? We give the answer to the first question "sufficient conditions" by another construction called CwF+A. Where the 'A' stands for adjunction.

Definition 2 (CwF+A)**.** A CwF+A is a CwF with an adjuntion $\mathsf{L} \dashv \mathsf{R}$ on contexts along with a lifting of the right adjoint to families, i.e.

$$\frac{\Gamma \vdash A}{\mathsf{R}\Gamma \vdash \mathsf{R}A} \qquad \frac{\Gamma \vdash t : A}{\mathsf{R}\Gamma \vdash \mathsf{R}t : \mathsf{R}A} \qquad \mathsf{R}A[\mathsf{R}\gamma] = \mathsf{R}(A[\gamma]) \qquad \mathsf{R}t[\mathsf{R}\gamma] = \mathsf{R}(t[\gamma])$$

Along with an isomorphism $\nu_{\Gamma,A} : \mathsf{R}\Gamma.\mathsf{R}A \to \mathsf{R}(\Gamma.A)$

$$\mathsf{R}\mathsf{p}_A \circ \nu_{\Gamma,A} = \mathsf{p}_{\mathsf{R}A} \qquad (\mathsf{R}\mathsf{q}_A)[\nu_{\Gamma,A}] = \mathsf{q}_{\mathsf{R}A} \qquad \nu_{\Gamma,A} \circ \langle \mathsf{R}\gamma, \mathsf{R}(a) \rangle = \mathsf{R}\langle \gamma, a \rangle$$

Lemma 2. *A CwF+A is a CwDRA and the two notions are equivalence if the CwF is democractic (i.e. where every context comes from a type [1]).*

Moreover we show how one can obtain a CwDRA from a cartesian closed category with adjoint endofunctors.

Lemma 3. *If $\mathscr{C}$ is a cartesian closed category and $L \dashv R$ are adjoint endofunctors on $\mathscr{C}$, then the Giraud construction [4] has the structure of CwDRA.*

References

[1] Pierre Clairambault and Peter Dybjer. *The Biequivalence of Locally Cartesian Closed Categories and Martin-Löf Type Theories*, pages 91–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[2] Ranald Clouston. Fitch-style modal lambda calculi. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 258–275, Cham, 2018. Springer International Publishing.

[3] Peter Dybjer. *Internal type theory*, pages 120–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[4] P. L. Lumsdaine and M. A. Warren. The local universes model: An overlooked coherence construction for dependent type theories. *ACM Trans. Comput. Logic*, 16(3):23:1–23:31, July 2015.

[5] Bassel Mannaa and Rasmus Ejlers Møgelberg. The clocks they are adjunctions: Denotational semantics for clocked type theory. 2018.

[6] Andrew M. Pitts, Justus Matthiesen, and Jasper Derikx. A dependent type theory with abstractable names. *Electronic Notes in Theoretical Computer Science*, 312:19 – 50, 2015. Ninth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2014).

Extensional and Intensional Semantic Universes: A Denotational Model of Dependent Types

Valentin Blot[1] and Jim Laird[2]

[1] LRI, Université Paris Sud, CNRS
Université Paris-Saclay, France
[2] Department of Computer Science
University of Bath, UK

Intuitionistic dependent type theory has been proposed as a foundation for constructive mathematics [9], within which proofs correspond to functional programs with dependent types which precisely specify their properties [4]. It is the basis for proof assistants and dependently typed programming languages such as Coq, Agda and Idris which exploit this correspondence. Its denotational semantics is therefore of interest — both for underpinning these logical foundations (cf. Martin-Löf's meaning-explanations for typing judgments [8]) — and in formulating and analysing new type systems — witness, for example, the role of the groupoid model [6] in the development of homotopy type theory. Most attention has been on models which are extensional in character (in particular, validating the principle of function extensionality).

Game semantics is also a foundational theory, describing the meaning of proofs and programs *intensionally* — i.e. how, rather than what, they compute — in terms of a dialogue between two players. With related models such as concrete data structures [5] it has been used to give interpretations of many programming languages and logical systems. These models are distinguished by desirable properties, notably, *full abstraction* [2, 7] and *full completeness* [1], but also connections to resource-sensitive computation and linear logic, direct representations of effectful computation, and the possibility of extracting computational content. Extending game semantics to dependent type theory is therefore a natural objective. It is also challenging, with little progress in this direction until recently [3]. Arguably, one source of difficulty is that the intensional representation of terms as strategies, which progressively reveal themselves by interaction, does not extend to types. This may reflect an intuition that types are static specifications and programs are more dynamic computational objects, but raises the question — how can one depend on the other? A related problem is: how can we interpret types themselves as terms of some special type (i.e. a *universe*) — a principle from which dependent type theory derives much of its expressive power — if the meanings of types and terms are defined in different ways?

We present solutions to these two problems, in the form of a new type theory, and a denotational semantics and categorical model for it. They are based on two "semantic universes" — intensional and extensional — of terms and types (or, more precisely, type and term formation judgments). Each of these has its own dependent type theory, and one can lift judgments from the intensional world to the extensional one — a form of cumulativity (corresponding to sending a program to the function it computes) — while the extensional universe contains a type of intensional types, so that type-families and type-operators can be represented as terms at this type.

Terms and types are interpreted as sequential algorithms and concrete data structures of a generalized form in the intensional universe, and as stable dependent functions and event domains in the extensional one. The concrete data structures themselves form an event domain, with which we may interpret an (extensional) universe type of (intensional) types.

The main technical challenge consists in the intensional interpretation of dependent types. Unfolding play in a dependent game constrains and enables future moves both explicitly and implicitly. We can capture this precisely by representing it as a *stable* function into the event domain of concrete data structures; we use its trace to define dependent product and sum constructions as it captures precisely how unfolding moves combine with the dependency to shape the possible interaction in the game, forming a bridge between the intensional and extensional worlds. Since each strategy computes a stable function on the states of a concrete data structure, we can lift typing judgements from the intensional to the extensional setting and interpret the cumulativity of our universes, giving an expressive type theory with recursively defined type families and type operators.

We define an operational semantics for intensional terms, giving a functional programming language based on our type theory, and prove that our semantics for it is computationally adequate. Adding a non-local control operator on intensional terms demonstrates that our type system can accommodate effectful computation, as well as leading to a *full abstraction* result for our model. Since it allows recursively defined type families, our type theory is very expressive but also partial, containing divergent programs at every type. By restricting to hereditarily total strategies we give a *fully complete* semantics of the recursion-free fragment. Additionally, our model enjoys an interesting interpretation of identity types. We leave the semantics of a more expressive total theory (i.e. with inductive rather than recursive types) as future work.

Acknowledgments

This research was supported by UK EPSRC grant EP/K037633/1, and by the Labex Digi-Cosme (project ANR11LABEX0045DIGICOSME) operated by ANR as part of the program "Investissements d'Avenir" Idex ParisSaclay (ANR11IDEX000302).

References

[1] Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543–574, 1994.

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full Abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[3] Samson Abramsky, Radha Jagadeesan, and Matthijs Vákár. Games for dependent types. In *42nd International Colloquium on Automata, Languages, and Programming*, pages 31–43. Springer, 2015.

[4] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. http://www.e-pig.org/downloads/ydtm.pdf, 2005.

[5] Gérard Berry and Pierre-Louis Curien. Sequential Algorithms on Concrete Data Structures. *Theoretical Computer Science*, 20(3):265–321, 1982.

[6] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-Five Years of Constructive Type Theory, Proceedings of a Congress held in Venice, October 1995*, Oxford Logic Guides, pages 83–111. Oxford University Press, 1998.

[7] Martin Hyland and Luke Ong. On Full Abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, 2000.

[8] Per Martin-Löf. Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*, volume 104, pages 153–175. Elsevier, 1982.

[9] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory - Lecture Notes. Bibliopolis, 1984.

Typing the Evolution of Variational Software

Luís Afonso Carvalho[1,2], João Costa Seco[1,2], and Jácome Cunha[1,3]

[1] NOVA LINCS
[2] Universidade NOVA de Lisboa
la.carvalho@campus.fct.unl.pt
joao.seco@fct.unl.pt
[3] Universidade do Minho
jacome@di.uminho.pt

Maintaining multiple versions of a software system is a laborious and challenging task, which is many times a strong requirement of the software development process. Such hurdle is justified by needs of backward compatibility with libraries or existence of legacy equipment with particular constraints. It is also an intrinsic requirement of software product lines that target multiple target platforms, service, or licensing levels [7].

A crucial example of a high variability context is an operating system where hundreds of variants need to be maintained to cope with all the different target architectures [1]. We find another important example in mobile applications, where server and client code need to be updated in sync to change structure of the interface or the semantics of webservices. However, it is always the case that older versions of server code must be maintained to support client devices that are not immediately updated. The soundness of a unique and common code corpus demands a high degree of design and programming discipline [8], code versioning, branching and merging tools [12], and sophisticated management methods [11, 9]. For instance, in loosely-coupled service-oriented architectures, where the compatibility guaranties between modules are almost non-existent, special attention is needed to maintain the soundness between multiple versions of service end-points (cf. Twitter API [13]).

Another issue regarding variability is the evolution of software. Arguably, existing language-based analysis tools for service orchestrations do not really account for evolution [14]. Nevertheless, there are other language- and type-based approaches that focus on dynamic reconfiguration and evolution of software [3, 4], hot swapping of code [10], and variability of software [5], that complement the evolution process with tools, and ensure that each version is sound. However, related versions of a software system usually share a significant amount of code, and there are no true guaranties of the sound co-existence of versions and sound transitions between versions at runtime. Such a need is relevant for monolithic software that must provide different versions in the same code base, and it is crucial in the context of service-based architectures. We have presented prior work to check the soundness of service APIs and the runtime transition between versions [2]. However, special hand crafted code was needed to maintain the semantic coherence of the versions of the state. Hence, we believe that the potential impact of a language-based tool supporting variability and a sound co-existence of versions is very high. By checking incremental evolution development it provides gains in safety and increases developer productivity.

Our approach is thus to provide a lightweight formal platform to solve the problem of multiplicity of code versions, while ensuring that the correct state transformations are executed when crossing contexts from one version to another. Our approach is a generalization of the main idea in [2] that keeps all versions well-typed at one given time. We consider one source file containing the code for all versions, and analysed as a whole. Versions and transitions between versions are made explicit in this model, as to represent code evolution steps. This code base is an analogy for a view over the entire history of a versioned code repository. Such a view

can be navigated with the help of a smart development environment that allows a developer to navigate in time, and identify errors in the evolution process.

We extend Featherweight Java (FJ) [6] with a type discipline that ensures that the evolution of state and functionality is captured and analysed. In a versioned FJ program, each element of a class is declared in a specific version context, and each expression is typed and executed with relation to a given version. Special key versions are used to mark state snapshots, where state variables and method types can change. Regular versions allow for the implementation of methods to be changed while maintaining their signature. Class constructors are used to define typed lenses between versions, declaring how an object (state) is legally translated from one version to another. Version contexts are tracked and transitions are only possible if there is a declared state transition. Any illegal version context crossing is dimmed as a typing error.

Such a type-based approach to the problem of maintaining multiple versions of a code base paves the ground for software construction and analysis tools that operate on main-stream languages and supporting runtime environments. Standard subject reduction results ensure that the ecosystem of versions is well formed and that any "view" on the code base is sound.

This work is supported by NOVA LINCS UID/CEC/04516/2013, COST CA15123 - EUTYPES and FC&T Project CLAY - PTDC/EEI-CTP/4293/2014.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *International Conference on Automated Software Engineering*, 2014.

[2] João Campinhos, João Costa Seco, and Jácome Cunha. Type-safe evolution of web services. In *Workshop on Variability and Complexity in Software Design, VACE@ICSE 2017*, 2017.

[3] João Costa Seco and Luís Caires. Types for dynamic reconfiguration. In *Symposium on Programming Programming Languages and Systems (ESOP 2006)*.

[4] Miguel Domingues and João Costa Seco. Type Safe Evolution of Live Systems. In *Workshop on Reactive and Event-based Languages & Systems (REBLS'15)*, 2015.

[5] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, 2011.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[7] Software Engineering Institute. Software product lines. www.sei.cmu.edu/productlines, 2016-8-16.

[8] Piotr Kaminski, Marin Litoiu, and Hausi Müller. A design technique for evolving web services. In *Conf. of the Center for Advanced Studies on Collaborative Research*.

[9] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis feasibility study. Technical Report CMU/SEI-90-TR-021, SE Institute, Carnegie Mellon University, 1990.

[10] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Dsu for java on a stock jvm. *SIGPLAN Not.*, 49(10):103–119, October 2014.

[11] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[12] Nayan B. Ruparelia. The history of version control. *SIGSOFT Softw. Eng. Notes*, 35(1):5–9, January 2010.

[13] Twitter Inc. Twitter API. https://developer.twitter.com.

[14] Hugo T. Vieira, Luís Caires, and João C. Seco. The conversation calculus: A model of service-oriented computation. In *European Conference on Programming Languages and Systems (ESOP 2008)*.

Polymorphic Gradual Typing: A Set-Theoretic Perspective

Giuseppe Castagna[1], Victor Lanvin[1], Tommaso Petrucciani[1,2], and Jeremy G. Siek[3]

[1] Université Paris Diderot, France [2] Università di Genova, Italy [3] Indiana University, USA

The aim of this work is to combine gradual typing, as introduced by Siek and Taha [3], with polymorphic union and intersection types as defined in the semantic subtyping approach [2].

Semantic subtyping is a technique by Frisch et al. [2] to define a type theory for union, intersection, and negation type connectives, in particular in the presence of higher-order functions. It consists in defining a semantic interpretation of types as sets (e.g., sets of values of some language) and then defining the subtyping relation as set-containment of the interpretations, whence the name of *set-theoretic types*. The advantage is that, by definition, types satisfy natural distribution laws (e.g., $(t \times s_1) \vee (t \times s_2)$ and $t \times (s_1 \vee s_2)$ are equivalent and so are $(s \to t_1) \wedge (s \to t_2)$ and $s \to (t_1 \wedge t_2)$).

Gradual typing is a recent approach that combines the safety guarantees of static typing with the flexibility and development speed of dynamic typing [3]. The idea behind it is to introduce an *unknown* type, often denoted by "`?`", used to inform the compiler that additional type checks may need to be performed at run time. Programmers can add type annotations to a program *gradually* and control precisely how much checking is done statically versus dynamically. The typechecker ensures that the parts of the program that are typed with *static types*—i.e., types that do not contain "`?`"—enjoy the type safety guarantees of static typing (well-typed expressions never get stuck), while the parts annotated with *gradual types*—i.e., types in which `?` occurs—enjoy the same property modulo the possibility to fail on some dynamic type check.

In this work we explore a new idea to interpret gradual types, namely, that the unknown type `?` acts like a type variable, but a peculiar one since each occurrence of `?` can be substituted by a different type. More precisely: a semantics of gradual types can be given by considering *each occurrence of* `?` *as a placeholder for a possibly distinct type variable*. We believe this idea to be the essence of gradual typing, and we formalize it by defining an operation of *discrimination* (denoted by $\odot$) which replaces each occurrence of `?` in a gradual type by a type variable.

Discrimination is the cornerstone of our semantics for gradual types: by applying discrimination we map a gradual type into a polymorphic set-theoretic type; then we use the semantic interpretation of the latter into a set, to interpret, indirectly, our initial gradual type. We use this semantic interpretation to revisit some notions from the gradual typing literature: we restate some of them, make new connections between them, and introduce new concepts. In particular, we use discrimination to define two preorders on gradual types: the *subtyping relation* (by which $\tau_1 \leq \tau_2$ implies that an expression of type τ_1 can be safely used where one of type τ_2 is expected) and the *materialization relation* ($\tau_1 \preccurlyeq \tau_2$ iff τ_2 is more precise—i.e., it has the same form but fewer occurrences of `?`—than τ_1). Using these preorders, we can define a gradual type system in a declarative form where terms are typed with standard rules (e.g., those of Hindley-Milner type systems) plus two non-syntax-directed rules corresponding to the two relations: subsumption (for subtyping) and materialization. The simplicity of this extension contrasts with current literature where gradual typing is obtained by embedding tests of consistency or of consistent subtyping (two non-transitive relations) in elimination rules.

Finally, our formalization partially brings to light the logical meaning of the cast language, the target language used by the compiler to add the dynamic type checks that ensure type safety. In our framework, expressions of the cast language encode (modulo the use of subsumption) the type derivations for the gradually-typed language. As such, the cast language looks like the missing ingredient in the Curry-Howard isomorphism for gradual typing disciplines. An intriguing direction for future work is to study the logic associated to these expressions.

A glimpse of formalization. We define subtyping on gradual types by *discrimination*, that is, by converting occurrences of ? to type variables to obtain static types (i.e., types without ?). Then, we reuse the existing semantic subtyping relation for polymorphic set-theoretic static types. That is, we define subtyping as

$$\tau_1 \leq \tau_2 \stackrel{\text{def}}{\iff} \exists t_1 \in \odot(\tau_1), t_2 \in \odot(\tau_2).\, t_1 \leq_t t_2$$

where $\odot(\tau)$ is the set of the possible discriminations of τ and where $\leq_t$ is the subtyping relation on polymorphic set-theoretic types defined in [1] (we use the meta-variables τ and t for gradual and static types, respectively). It turns out that in order to check subtyping, it is not necessary to consider all the possible discriminations of the two types. It suffices to check that the relation holds when we replace in both types all covariant occurrences of ? by one variable and all contravariant occurrences by a second variable different from the first (this property holds only for deciding subtyping and not, say, for typing: functions of type $? \to ? \to \mathsf{Int}$ and of type $\alpha \to \alpha \to \mathsf{Int}$ are completely different beasts).

The *materialization* relation on gradual types $\tau_1 \preccurlyeq \tau_2$ is defined as follows:

$$\tau_1 \preccurlyeq \tau_2 \stackrel{\text{def}}{\iff} \exists t_1 \in \odot(\tau_1). \exists \theta.\, t_1\theta = \tau_2$$

where θ is a substitution mapping the variables inserted by discrimination to gradual types. It turns out that this new definition characterizes the inverse relation of the "precision" relation of [4], it extends it to set-theoretic types and generalizes it since it is (type) syntax agnostic. We use these typing rules:

$$\frac{\Gamma(x) = \forall \vec{\alpha}.\, \tau}{\Gamma \vdash x \colon \tau[\vec{t}/\vec{\alpha}]} \qquad \frac{\Gamma, x \colon t \vdash e \colon \tau}{\Gamma \vdash (\lambda x.\, e) \colon t \to \tau} \qquad \frac{\Gamma, x \colon \tau' \vdash e \colon \tau}{\Gamma \vdash (\lambda x \colon \tau'.\, e) \colon \tau' \to \tau} \qquad \frac{\Gamma \vdash e_1 \colon \tau' \to \tau \qquad \Gamma \vdash e_2 \colon \tau'}{\Gamma \vdash e_1\, e_2 \colon \tau}$$

$$\frac{\Gamma \vdash e_1 \colon \tau_1 \qquad \Gamma, x \colon \forall \vec{\alpha}.\, \tau_1 \vdash e_2 \colon \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \colon \tau}\ \vec{\alpha} \sharp \Gamma \qquad [\leq]\ \frac{\Gamma \vdash e \colon \tau'}{\Gamma \vdash e \colon \tau}\ \tau' \leq \tau \qquad [\preccurlyeq]\ \frac{\Gamma \vdash e \colon \tau'}{\Gamma \vdash e \colon \tau}\ \tau' \preccurlyeq \tau$$

The first five rules are standard rules for Hindley-Milner type systems, with the subtlety that function parameters can be assigned gradual types only if they are explicitly annotated.

The type system is declarative insofar as the two key relations on gradual types, subtyping and materialization, are added to the system by two specific non-syntax-directed rules, $[\leq]$ and $[\preccurlyeq]$, stating that these relations can be used in any context. This is a novelty and one of the contributions of our work. Hitherto, gradual typing was obtained by inserting checks in elimination rules; although this describes the algorithmic aspects of the implementation, it hides the logical meaning of "graduality". Rule $[\leq]$ is standard (but uses a new subtyping relation), and states that the type of every well-typed expression can be subsumed to a supertype. Rule $[\preccurlyeq]$ is new (but uses a relation already existing in the literature on gradual types) and is the counterpart of subsumption for the precision relation. It states that we can subsume every well-typed expression to a more precise type obtained by replacing some occurrences of ? by a gradual type: this is a declarative way to embed consistency checks in the type system.

[1] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP '11*, 2011.

[2] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4), 2008.

[3] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of Scheme and Functional Programming Workshop*, 2006.

[4] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32, 2015.

Vectors are records, too

Jesper Cockx[1], Gaëtan Gilbert[2], and Nicolas Tabareau[2]

[1] Gothenburg University, Sweden
[2] INRIA, France

When talking about dependently typed programming languages such as Coq and Agda, it is traditional to start with an example involving vectors, i.e. length-indexed lists:

$$\begin{array}{l}\mathsf{data}\ \mathsf{Vector}\ (A : \mathsf{Set}) : (n : \mathbb{N}) \to \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{nil} \quad : \quad \mathsf{Vector}\ A\ \mathsf{zero} \\ \quad \mathsf{cons} \quad : \quad (m : \mathbb{N})(x : A)(xs : \mathsf{Vector}\ A\ m) \to \mathsf{Vector}\ A\ (\mathsf{suc}\ m)\end{array} \tag{1}$$

This definition of vectors as an indexed family of datatypes is very intuitive: we take the definition of lists and ornament them with their length. Alternatively, we can also define vectors by recursion on the length:

$$\begin{array}{lcl}\mathsf{Vector} : (A : \mathsf{Set})(n : \mathbb{N}) \to \mathsf{Set} & & \\ \mathsf{Vector}\ A\ \mathsf{zero} & = & \top \\ \mathsf{Vector}\ A\ (\mathsf{suc}\ n) & = & A \times \mathsf{Vector}\ A\ n\end{array} \tag{2}$$

This transformation of an indexed family of inductive datatypes (or *indexed datatype* for short) into a recursive definition has a number of benefits:

- Vector inherits η-laws from the record types $\top$ and $A \times B$: every vector of length zero is definitionally equal to $\mathsf{tt} : \top$, and every vector $xs : \mathsf{Vec}\ A\ (\mathsf{suc}\ n)$ is definitionally equal to the pair $(\mathsf{fst}\ xs, \mathsf{snd}\ xs)$ where $\mathsf{fst}\ xs : A$ and $\mathsf{snd}\ xs : \mathsf{Vector}\ A\ n$.
- We get the forcing and detagging optimizations from Brady et al. (2004) for free: we do not have to store the length of a vector, and not even whether it is a nil or a cons.
- There are no restrictions on the sorts of the types of forced indices; they can be in a bigger sort than the datatype itself. In particular, this allows us to define indexed datatypes in a proof-irrelevant universe such as Prop, as long as the constructor can be uniquely determined from the indices and all non-forced constructor arguments are in the proof-irrelevant universe themselves.
- The recursive occurrences of the datatype do not have to be strictly positive: they only have to use a structurally smaller index. This allows us to define stratified types as in Beluga (Pientka, 2015).

While this transformation works for vectors, it is not possible for all datatypes. For example, the recursive definition of $\mathbb{N}$ by the equation $\mathbb{N} = \top \uplus \mathbb{N}$ is invalid since it is not terminating. This explains why we cannot allow η-laws for all datatypes.

As another counterexample, consider the datatype $\mathsf{Image}\ (A\ B : \mathsf{Set})(f : A \to B) : B \to \mathsf{Set}$ with one constructor $\mathsf{image} : (x : A) \to \mathsf{Image}\ A\ B\ f\ (f\ x)$. Image cannot be defined by pattern matching on $y : B$ since $f\ x$ is not a pattern. We can instead transform the index into a parameter by introducing an equality proof: $\mathsf{Image}\ A\ B\ f\ y = \Sigma_{(x:A)}(f\ x \equiv_B y)$. This transformation removes the non-pattern index and hence allows us to match against an element of $\mathsf{Image}\ A\ B\ f\ u$ even when u is not a variable. On the other hand, this transformation does

not enable us to have large indices: we cannot define Image in Prop since both A and $f\ x \equiv_B y$ have to fit in the sort of Image.

Both these transformations for removing the indices from a datatype definition as described above are well known, but so far the only way to get their benefits was to apply them by hand. This means that we also have to define terms for the constructors and the elimination principle ourselves, and we cannot rely on built-in support for indexed datatypes such as dependent pattern matching.

We present a fully automatic and general transformation of an indexed datatype to an equivalent definition of a type as a case tree. This transformation generates not just the type itself but also terms for the constructors and the elimination principle. It exposes eta laws for datatypes when there is only a single possible constructor for the given indices, and removes non-pattern indices by introducing equality proofs as new constructor arguments.

Our transformation is similar to the elaboration of dependent pattern matching (Goguen et al., 2006). It uses pattern matching on the indices where it can, and introduces equality proofs where it must. First we elaborate the datatype declaration to a case tree where each internal node indicates a case split on one of the indices, and each leaf node contains some (possibly zero) telescopes for the arguments of each constructor. For Vector we get:

$$\mathsf{Vector} = \lambda A, n.\ \mathsf{case}_n \left\{ \begin{array}{lcl} \mathsf{zero} & \mapsto & () \\ \mathsf{suc}\ m & \mapsto & (x : A)(xs : \mathsf{Vector}\ A\ m) \end{array} \right\} \tag{3}$$

Any non-constructor patterns and non-linear variables are dealt with by replacing them with fresh variables and introducing equality types on the right-hand side. For Image we get:

$$\mathsf{Image} = \lambda y.\ (x : A)(p : f\ x \equiv_B y) \tag{4}$$

Once we have a case tree, it is straightforward to construct a definition for the datatype itself, as well as for the constructors and the eliminator.

Our approach is similar to the notion of case-splitting datatypes of Dagand and McBride (2014), but we do not require any annotations from the user. One could however imagine extending our approach with some user annotations to guide the elaboration process, similar to the inaccessible patterns from dependent pattern matching.

For a long time datatypes have been discriminated against by refusing to give them eta equality and restricting the sort of their indices. We say: no more! Let vectors be records, too.

References

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, 2004.

Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of functional programming*, 24(2-3):316–383, 2014.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*. 2006.

Brigitte Pientka. *Beluga 0.8.2 Reference Guide*, 2015. URL `http://complogic.cs.mcgill.ca/beluga/userguide2/userguide.pdf`.

Mailbox Types for Unordered Interactions

Ugo de'Liguoro and Luca Padovani

Università di Torino, Dipartimento di Informatica, Torino, Italy

Message passing is a key mechanism used to coordinate concurrent processes. The order in which a process consumes messages may coincide with the order in which they arrive at destination (*ordered processing*) or may depend on some intrinsic property of the messages themselves, such as their priority, their tag, or the shape of their content (*out-of-order or selective processing*). Ordered message processing is common in networks of processes connected by point-to-point *channels*. Out-of-order message processing is common in networks of processes using *mailboxes*, into which processes concurrently store messages and from which one process selectively receives messages. This communication model is typically found in the various implementations of actors such as Erlang, Scala and Akka, CAF and Kilim. Out-of-order message processing adds further complexity to the challenging task of concurrent and parallel application development: storing a message into the wrong mailbox or at the wrong time, forgetting a message in a mailbox, or relying on the presence of a particular message that is not guaranteed to arrive in a mailbox are programming mistakes that are easy to do and hard to detect without adequate support from the language and its development tools. Type-based static analysis techniques can help developers in detecting mistakes like these. In this respect, we make the following contributions:

- We introduce *mailbox types*, a new kind of behavioral types that allow us to describe mailboxes subject to selective message processing. Incidentally, mailbox types also provide precise information on the size of mailboxes that may lead to valuable code optimizations.
- We define a mailbox type system for the *mailbox calculus*, a mild extension of the asynchronous π-calculus featuring tagged messages, selective inputs and first-class mailboxes.
- We prove three properties of well-typed processes: the absence of failures due to unexpected messages (*mailbox conformance*); the absence of pending activities and messages in irreducible processes (*deadlock freedom*); for a non-trivial class of processes, the guarantee that every message can be eventually consumed (*junk freedom*).

Several behavioral type systems that enforce safety properties of communicating processes have already been studied. In particular, session types [8] have proved to be an effective formalism for the enforcement of communication protocols and have been applied to a variety of programming paradigms and languages, including those using mailbox communications. However, session types are built using connectives expressing choice and sequential protocol composition and therefore are specifically designed to address ordered interactions over channels. In contrast, mailbox types are elements of a *commutative Kleene algebra* [4] that embody the unordered nature of mailboxes and enable the description of mailboxes concurrently accessed by several processes, abstracting away from the state and behavior of the objects/actors/processes using these mailboxes. The fact that a mailbox may have different types during its lifetime is entirely encapsulated by the typing rules and not apparent from mailbox types themselves.

In the pure actor model [7, 1], each actor owns a single mailbox and the only synchronization mechanism is message reception from such mailbox. The practice of programming with actors, however, is not that simple. For example, it is a known fact that the implementation of complex coordination protocols in this model is challenging. These difficulties have led programmers to mix

the actor model with different concurrency abstractions [9, 11], to extend actors with controlled forms of synchronization [12] and to consider actors with multiple/first-class mailboxes [6, 10, 3]. In fact, popular implementations of the actor model feature disguised instances of multiple/first-class mailbox usage, even if they are not explicitly presented as such: in Akka, the messages that an actor is unable to process immediately can be temporarily stashed into a different mailbox [6]; in Erlang, hot code swapping implies transferring at runtime the input capability on a mailbox from a piece of code to a different one [2]. As a consequence, static analysis techniques targeting the pure actor model fall short at addressing real-world programs, which tend to use a richer set of concurrency abstractions and to mix different concurrency models. The mailbox calculus is general enough to subsume the actor model and, additionally, to model a broader range of systems with a dynamic communication topology and a varying number of processes mixing different concurrency abstractions (including locks, futures, multiple and first-class mailboxes).

Besides the potential applications of this work to the analysis of communicating processes, mailbox types and the type system for the mailbox calculus exhibit intriguing analogies with (polarized) linear logic and its proof system. We plan to investigate the exact relationship between our type system and linear logic in future work.

Full details on our type system and formal proofs of the results can be found in the technical report [5].

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

[3] Arghya Chatterjee, Branko Gvoka, Bing Xue, Zoran Budimlic, Shams Imam, and Vivek Sarkar. A distributed selectors runtime system for java applications. In *Proceedings of PPPJ'16*, pages 3:1–3:11. ACM, 2016.

[4] John Conway. *Regular Algebra and Finite Machines*. William Clowes & Sons Ltd, 1971.

[5] Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In *Proceedings of ECOOP'18*, 2018. Technical report available at `http://arxiv.org/abs/1801.04167`.

[6] Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of AGERE! 2012*, pages 1–6. ACM, 2012.

[7] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of IJCAI'73*, pages 235–245. William Kaufmann, 1973.

[8] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, 2016.

[9] Shams Mahmood Imam and Vivek Sarkar. Integrating task parallelism with actors. *SIGPLAN Notices*, 47(10):753–772, 2012.

[10] Shams Mahmood Imam and Vivek Sarkar. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of AGERE! 2014*, pages 1–14. ACM, 2014.

[11] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proceedings of ECOOP'13*, LNCS 7920, pages 302–326. Springer, 2013.

[12] Carlos A. Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices*, 36(12):20–34, 2001.

A Simpler Undecidability Proof for System **F** Inhabitation

Andrej Dudenhefner[1] and Jakob Rehof[1]

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

Polymorphic λ-calculus (also known as Girard's "system **F**" [3] or $\lambda 2$ [2]) is directly related to intuitionistic second-order propositional logic (IPC_2) via the Curry–Howard isomorphism (for an overview see [5]). In particular, provability in the implicational fragment of IPC_2 (is a given formula an IPC_2 theorem?) corresponds to inhabitation in system **F** (given a type, is there a term having that type in system **F**?).

Provability in IPC_2 was shown by Löb to be undecidable [4]. The proof itself is by reduction from provability in first-order predicate logic via a semantic argument. Since the original proof is heavily condensed (14 pages), Arts in collaboration with Dekkers provided a fully unfolded argument [1] (approx. 50 pages) reconstructing the original proof. Later, Sørensen and Urzyczyn developed a different, syntax oriented proof showing undecidability of inhabitation in system **F** [5, Section 11.6] (6 pages, moderately condensed).

In order to show undecidability of provability in IPC_2, each of the above approaches embeds first-order predicate logic into IPC_2. However, if one is solely interested in a concise and rigorous proof (e.g. for formalization or didactics), then there is no need for a full embedding. In fact, we can pursue a different approach to show undecidability of inhabitation in system **F**. In this extended abstract, we sketch a reduction from solvability of Diophantine equations (is there an integer solution to $P(x_1, \ldots, x_n) = 0$ where P is a polynomial with integer coefficients?) to inhabitation in system **F**. We argue that, compared to the previous approaches, the sketched reduction is more accessible for formalization and more comprehensible for didactic purposes.

First, let us fix some notation. Let *type variables* be ranged over by $a, b, c, \ldots$, we define *polymorphic types* ranged over by $\sigma, \tau, \rho, \ldots$ as

$$\sigma, \tau, \rho ::= a \mid \sigma \to \tau \mid \forall a.\sigma$$

Let $M, N, \ldots$ be ranged over Church-style polymorphic λ-calculus terms defined as

$$M, N ::= x \mid (M\, N) \mid (\lambda x : \sigma.M) \mid (\Lambda a.M) \mid M\, \tau$$

Let $\Delta = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ denote finite *type environments*. Typing rules of system **F** deriving *judgements* $\Delta \vdash M : \sigma$ are as usual [5, Section 11.2].

As a starting point, we use the following Problem 1, which is undecidable by reduction (routine polynomial decomposition) from solvability of Diophantine equations.

Problem 1. *Given a set* $\mathsf{A} = \{\mathfrak{e}_1, \ldots \mathfrak{e}_l\}$ *of constraints over* $\mathcal{V} = \{a_1, \ldots, a_n\}$ *where each* $\mathfrak{e} \in \mathsf{A}$ *is of shape either* $a \doteq 1$ *or* $a \doteq b + c$ *or* $a \doteq b \cdot c$ *for some* $a, b, c \in \mathcal{V}$*, does there exist a substitution* $\zeta : \mathcal{V} \to \mathbb{N}$ *that satisfies* A*?*

Reducing Problem 1 to inhabitation in system **F** it suffices to axiomatize natural number addition and multiplication. Let us fix an instance of A of Problem 1 over variables $\{a_1, \ldots, a_n\}$. In the remainder of this long abstract we sketch the construction of a type environment Δ^{A} and type τ^{A} such that A has a solution iff there exists a term M such that $\Delta^{\mathsf{A}} \vdash M : \tau^{\mathsf{A}}$.

To simplify notation, let us define the following types (where $\dagger, \mathbf{1}, \ldots$ are standard type variables)

$$\dagger\,\sigma = \sigma \to \dagger \qquad U(\sigma) = (\dagger\sigma \to \bullet_1) \to (\sigma \to \bullet_2) \to u$$
$$S(\sigma,\tau,\rho) = (\dagger\sigma \to \bullet_1) \to (\dagger\tau \to \bullet_2) \to (\dagger\rho \to \bullet_3) \to s$$
$$P(\sigma,\tau,\rho) = (\dagger\sigma \to \bullet_1) \to (\dagger\tau \to \bullet_2) \to (\dagger\rho \to \bullet_3) \to p$$
$$\overline{a \doteq 1} = P(\mathbf{1},\mathbf{1},a) \qquad \overline{a \doteq b + c} = S(b,c,a) \qquad \overline{a \doteq b \cdot c} = P(b,c,a)$$

Intuitively, the type variable $\mathbf{1}$ represents $1 \in \mathbb{N}$, the type $U(\sigma)$ signifies that σ is an element of a universe $\mathcal{U}$, and $S(\sigma,\tau,\rho)$ (resp. $P(\sigma,\tau,\rho)$) signifies that the sum (resp. product) of the two elements σ and τ is ρ. We axiomatize natural number arithmetic as follows

$$\Delta_{\mathbb{N}} = \{x_u : \forall a.\Big(U(a) \to \forall b.(U(b) \to S(a,\mathbf{1},b) \to P(b,\mathbf{1},b) \to \blacktriangle) \to \blacktriangle\Big),$$
$$x_s : \forall abcde.\Big(U(a) \to U(b) \to U(c) \to U(d) \to U(e) \to$$
$$S(a,b,c) \to S(b,\mathbf{1},d) \to S(c,\mathbf{1},e) \to (S(a,d,e) \to \blacktriangle) \to \blacktriangle\Big),$$
$$x_p : \forall abcde.\Big(U(a) \to U(b) \to U(c) \to U(d) \to U(e) \to$$
$$P(a,b,c) \to S(b,\mathbf{1},d) \to S(c,a,e) \to (P(a,d,e) \to \blacktriangle) \to \blacktriangle\Big), y_u^{U(\mathbf{1})} : U(\mathbf{1}), y_p^{P(\mathbf{1},\mathbf{1},\mathbf{1})} : P(\mathbf{1},\mathbf{1},\mathbf{1})\}$$

Type assumptions in $\Delta_{\mathbb{N}}$ encompass the following assertions about members of a universe $\mathcal{U}$

- $y_u^{U(\mathbf{1})}$ asserts that $\mathbf{1} \in \mathcal{U}$ and $y_p^{P(\mathbf{1},\mathbf{1},\mathbf{1})}$ asserts that $\mathbf{1} \cdot \mathbf{1} = \mathbf{1}$
- x_u asserts that for any $a \in \mathcal{U}$ there is $b \in \mathcal{U}$ such that $a + \mathbf{1} = b$ and $b \cdot \mathbf{1} = b$
- x_s asserts for $a,b,c,d,e \in \mathcal{U}$: if $a + b = c$, $b + \mathbf{1} = d$ and $c + \mathbf{1} = e$, then $a + d = e$
- x_p asserts for $a,b,c,d,e \in \mathcal{U}$: if $a \cdot b = c$, $b + \mathbf{1} = d$ and $c + a = e$, then $a \cdot d = e$

Let $\Delta^{\mathsf{A}} = \Delta_{\mathbb{N}} \cup \{x_{\mathsf{A}} : \forall a_1 \ldots a_n.\Big(U(a_1) \to \ldots \to U(a_n) \to \overline{\mathfrak{C}_1} \to \ldots \to \overline{\mathfrak{C}_l} \to \blacktriangle\Big)\}$ and $\tau^{\mathsf{A}} = \blacktriangle$. We are able to establish soundness (cf. Theorem 1) and completeness (cf. Theorem 2) of our encoding.

Theorem 1 (Soundness)**.** *If $\Delta^{\mathsf{A}} \vdash M : \tau^{\mathsf{A}}$ for some M, then A has a solution.*

Theorem 2 (Completeness)**.** *If A has a solution, then $\Delta^{\mathsf{A}} \vdash M : \tau^{\mathsf{A}}$ for some M.*

A rigorous proof of soundness by routine case analysis uses the key property of system **F** that any inhabited type is inhabited by some β-normal η-long Church-style term. The proof of completeness is by direct construction of an inhabitant for a given solution of A. A formalization of the above reduction is currently under development.

References

[1] T. Arts and W. Dekkers. Embedding first order predicate logic in second order propositional logic. *Master's thesis, University of Nijmegen*, 1992.

[2] H. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[3] J. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[4] M. H. Löb. Embedding first order predicate logic in fragments of intuitionistic logic. *J. Symb. Log.*, 41(4):705–718, 1976.

[5] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

A Comparative Analysis of Type-Theoretic Interpretations of Constructive Set Theories

Cesare Gallozzi

University of Leeds, UK

Abstract

This talk will present a comparative analysis of some interpretations of constructive set theories into type theories. The analysis will be carried out by factoring the interpretations via logic-enriched type theories, in this way the key principles used in the interpretations are isolated and can be compared.

The comparison is between on the one hand Aczel's interpretation of CZF into Martin-Löf type theory [1], and on the other hand the interpretations developed recently [4], [5] and [2] of Myhill's Constructive Set Theory into homotopy type theory.

1 Introduction and Motivation

In this talk we will consider two constructive set theories: Aczel's Constructive Zermelo-Fraenkel (CZF) and Myhill's Constructive Set Theory (CST). The only difference between the two is that CST includes the Replacement axiom and the Exponentiation axiom[1], whereas CZF has the Strong Collection and Subset Collection axioms which strengthen Replacement and Exponentiation, respectively.

In [1], Aczel developed an interpretation of CZF into Martin-Löf type theory in which formulas are interpreted using the propositions-as-types correspondence. This interpretation is referred as set-as-trees since nodes represent sets and edges the membership relation, so that the root represents the given set. This intuition is formalised by interpreting the universe of sets as the type $V := (Wx : U)x$.

Recently, new interpretations of CST into homotopy type theory have been developed in the HoTT book [5], by Gylterud [4], and by the author [2], all inspired by Aczel's interpretation.

These interpretations make use of homotopical notions such as homotopy levels, the Univalence axiom and Higher Inductive Types. In particular, [2] introduced a family of interpretations $[\![\cdot]\!]_{k,h}$, where k represents the homotopy level of the interpretation of sets and h the one of propositions. Among those, the interpretations that validate the axioms of CST are $[\![\cdot]\!]_{k,1}$ for $2 \leq k \leq \infty$.

The interpretations in the HoTT book, by Gylterud and the $[\![\cdot]\!]_{k,1}$ are equivalent (see [4] and [2]).

A natural question is then to relate and compare these equivalent interpretation of CST into homotopy type theory with Aczel's interpretation of CZF into Martin-Löf type theory.

Logic-enriched type theories provide a convenient and conceptually clear framework for this kind of analysis. These are type theories that in addition to a pure type-theoretic part have primitive formulas and judgments for formulas $\Gamma \vdash \phi_1, \ldots, \phi_n \Rightarrow \phi$. Free variables of formulas range over types, which can be quantified, e.g. $(\forall x : A)\phi(x)$, and the usual logical rules define connectives and quantifiers. The propositions-as-types correspondence is not a default feature but can be added easily.

[1] Given two sets A, B, there exists the set of functions B^A

2 Overview of the Results

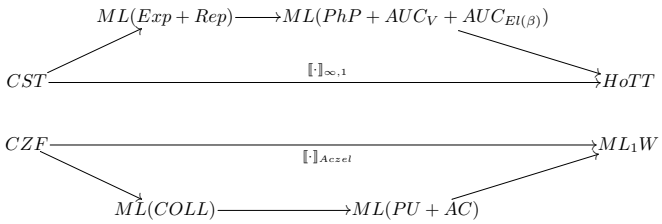
Among the equivalent interpretations of CST we will focus on $[\![\cdot]\!]_{\infty,1}$ which is the best suited for this kind of analysis.

The paper [3] provides an analysis of Aczel's interpretation. The interpretation is factored through a logic-enriched type theory consisting of Martin-Löf type theory with intuitionistic logic and two new rules: a propositions-as-types principle (PU), and the type-theoretic axiom of choice formulated in the language of the logic-enriched type theory (AC).

This talk will present a similar factorisation of the interpretation $[\![\cdot]\!]_{\infty,1}$ through a logic-enriched type theory with three new rules: a propositions-as-hpropositions principle (PhP) which strengthens (PU), and two rules that have the form of the Axiom of Unique Choice, one for the type of sets (AUC_V) and one for the terms of the type of sets ($AUC_{El(\beta)}$), both of them follow from (AC). These three rules are valid when interpreted in homotopy type theory and allow to validate respectively the Bounded Separation, Replacement and Exponentiation axioms of CST.

In [3] the interpretation of CZF into $ML(PU + AC)$ is factored further via another logic-enriched type theory with two Collection Rules ($COLL$) mirroring the two Collection axioms. Similarly, we will factor the interpretation of CST into $ML(PhP + AUC_V + AUC_{El(\beta)})$ via another logic-enriched type theory with Replacement and Collection rules (Rep) and (Exp) mirroring the Replacement and Exponentiation axioms. These two rules are consequences of the Collection Rules ($COLL$). Under the assumption of (PhP) the Replacement and Exponentiation Rules follow respectively from (AUC_V) and ($AUC_{El(\beta)}$).

The two factorisations follow the same structure very closely in the first two steps, but differ in the third one. In the case of CST the interpretation of the two axioms of unique choice into $HoTT$ is not obvious and rests on the equivalence between $[\![\cdot]\!]_{\infty,1}$ and Gylterud's interpretation, which in turn uses univalence and set-quotients. In summary, we obtain the following diagram.



References

[1] Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic colloquium*, volume 77, pages 55–66, 1978.

[2] Cesare Gallozzi. Homotopy type-theoretic interpretations of constructive set theories. PhD thesis, School of Maths, University of Leeds, in preparation, 2018.

[3] Nicola Gambino and Peter Aczel. The generalised type-theoretic interpretation of constructive set theory. *The Journal of Symbolic Logic*, 71(1):67–103, 2006.

[4] Håkon Robbestad Gylterud. From multisets to sets in hotmotopy type theory. *arXiv preprint arXiv:1612.05468*, 2016.

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

Normalisation for general constructive propositional logic

Herman Geuvers, Iris van der Giessen, and Tonny Hurkens

Radboud University & Technical University Eindhoven

Overview

In [2] we have developed a general method for deriving natural deduction rules from the truth table for a connective. The method applies to both constructive and classical logic. We have shown completeness with respect to Kripke semantics and we have defined the notion of *detour conversion* (or "cut") for the constructive connectives. We have also shown that for the well-known connectives, like $\vee$, $\wedge$, $\to$, the constructive rules we derive are equivalent to the natural deduction rules we know from Gentzen and Prawitz. However, they have a different shape (closer to the "general elimination rules" by Von Plato [4]), because we want all our rules to have a standard "format", to make it easier to define the notion of detour conversion in general.

In [3] we have analysed detour conversion in general for the constructive rules. Following the Curry-Howard isomorphism, we have given terms to deductions and we have studied detour conversion as term reduction. We have proven in [3] that reduction is weakly normalising for any set of constructive rules, where we consider the union of *detour conversion* and *permutation conversion*. In the present talk we will prove *strong normalisation* of the logic for any set of constructive rules, strengthening the weak normalisation result of [3].

Background

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

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

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

The rules are given in abbreviated form and it should be understood that all judgements can be used with an extended hypotheses set Γ.

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

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

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

These rules are all intuitionistically correct, as one can observe by inspection. In [2, 3] it has been shown that these can be "optimised" and be reduced to 2 elimination rules and 1 introduction rule: the well-known intuitionistic rules. It should be noted that for each connective, the rules are completely self-contained, so we don't need to explain one connective in terms of another.

Contribution

In [2, 3] we have defined the notions of *detour convertibility* (which we called *direct cut* in [2]): an introduction rule immediately followed by an elimination rule, and the notion of *permutation convertibility*, (which we called *direct cut* in [2]): an elimination rule followed by an elimination rule that may block a detour convertibility. These give rise to two ways of transfoming natural deductions: *detour conversions* and *permutation conversions*. In [3] we have proven that detour conversion is strongly normalising, that permutation conversion is strongly normalising and that the combination of both is weakly normalising. In the present talk we will strengthen this result and proof strong normalisation (SN) of the combination.

We will establish the SN result in the following way: We define a translation of our logic into a simple type theory with "parallel terms", $\lambda{\to}^{\mathrm{par}}$. (Reduction in our logic is not Church-Rosser, so a translation simply to $\lambda\to$ would not do.) The translation for formulas (types) is in double negation style, and the translation of proof terms is in CPS style, following De Groote [1]. The term translation is reduction preserving and we prove that $\lambda{\to}^{\mathrm{par}}$ is strongly normalising, thereby obtaining the result.

Inspired by the SN proof, we describe a general logic that has a weak form of negation, for which strong normalisation is relatively easy to prove. Any constructive logic for any set of connectives that we have described can be translated to this general logic, in such a way that detour conversion is preserved. The permutation conversions are preserved equationally, but not as reductions. The logic has a weak type of negation $\sim A$, which we can view as $A \to o$, where o is a basic proposition without further logical meaning. The formulas are the ones formed using the connectives of choice, φ, ψ, plus the ones of the form $\sim\varphi$, $\sim\sim\varphi$ and $\sim\sim\sim\varphi$. The rules are as follows.

$$\frac{\sim\sim\sim\varphi \quad \sim\sim\varphi}{o} \qquad \frac{\sim\sim\varphi \quad \sim\varphi}{o} \qquad \frac{\sim\varphi \quad \varphi}{o} \qquad \frac{\begin{array}{c}[\sim\sim\varphi]\\ \vdots\\ o\end{array}}{\sim\sim\sim\varphi} \qquad \frac{\begin{array}{c}[\sim\varphi]\\ \vdots\\ o\end{array}}{\sim\sim\varphi}$$

For every connective we have rules specific for that connective. Here we give the ones for $\wedge$.

$$\frac{\sim\sim P \quad \sim\sim Q}{P \wedge Q}\,\wedge\text{-in} \qquad \frac{\sim\sim\sim P}{\sim(P \wedge Q)}\,\wedge\text{-el}_1 \qquad \frac{\sim\sim\sim Q}{\sim(P \wedge Q)}\,\wedge\text{-el}_2$$

References

[1] Philippe de Groote. On the strong normalisation of intuitionistic natural deduction with permutation-conversions. *Inf. Comput.*, 178(2):441–464, 2002.

[2] Herman Geuvers and Tonny Hurkens. Deriving natural deduction rules from truth tables. In *ICLA*, volume 10119 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2017.

[3] Herman Geuvers and Tonny Hurkens. Proof terms for generalized natural deduction, accepted for publication at Types 2017, 2018. See http://www.cs.ru.nl/H.Geuvers/PUBS/ProofTermsNaturalDeduction.pdf.

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

Towards Probabilistic Reasoning about Typed Lambda Terms

Silvia Ghilezan[1,2], Jelena Ivetić[1], Simona Kašterović[1], Zoran Ognjanović[2], and Nenad Savić[3]

[1] University of Novi Sad, Novi Sad, Serbia
{gsilvia, jelenaivetic, simona.k}@uns.ac.rs
[2] Mathematical Institute SANU, Belgrade, Serbia
zorano@mi.sanu.ac.rs
[3] Institute of Computer Science, University of Bern, Switzerland
savic@inf.unibe.ch

Reasoning with uncertainty has gained an important role in logic, computer science, artificial intelligence and cognitive science. These applications urge for development of formal models which capture reasoning of probabilistic features. The general lack of compactness for probabilistic logics causes that one of the main proof-theoretical problems in this framework is to provide a strongly complete axiomatic system. Several infinitary logics have been introduced to deal with that issue, a detailed overview can be found in [5, 6]. Note that the term infinitary concerns the meta language only, i.e. the object language is countable, and formulas are finite, while only proofs are allowed to be infinite.

We will present the results of [3] and ongoing work that emerged from [3] and [2]. In [3] we have introduced a formal model $\mathsf{P}\Lambda_{\to}$ for reasoning about probabilities of simply typed lambda terms which is a combination of lambda calculus and probabilistic logic. The probabilistic logic that we use is LPP_2 logic, a detailed overview can be found in [6]. Our ideas for proving strong completeness are based on the ideas used in [6] for LPP_2 logic. The simply type assignment, which is sound and complete ([4]) with respect to the simple semantics (based on a concept of a term model) is used.

We have proposed ([3]) a syntax, Kripke-style semantics and an infinitary axiomatization for $\mathsf{P}\Lambda_{\to}$. The *language* of $\mathsf{P}\Lambda_{\to}$ consists of two sets of formulas, basic formulas and probabilistic formulas.

$$\mathsf{For}_{\mathsf{P}\Lambda_{\to}} = \mathsf{For}_{\mathsf{B}} \cup \mathsf{For}_{\mathsf{P}}.$$

Basic formulas, denoted by $\mathsf{For}_{\mathsf{B}}$, are all lambda statements of the form $M : \sigma$, where M is lambda term and σ is a simple type or statements of the same form connected with Boolean connectives.

$$\mathsf{For}_{\mathsf{B}} \quad \alpha ::= M : \sigma \mid \alpha \wedge \alpha \mid \neg\alpha.$$

Basic probabilistic formulas are formulas of the form $P_{\geq s}\alpha$, where α is a basic formula and $s \in [0,1] \cap \mathbb{Q}$. The set of all probabilistic formulas, denoted by $\mathsf{For}_{\mathsf{P}}$, is the smallest set containing all basic probabilistic formulas which is closed under Boolean connectives.

$$\mathsf{For}_{\mathsf{P}} \quad \phi ::= P_{\geq s}\alpha \mid \phi \wedge \phi \mid \neg\phi.$$

In our language nested probabilistic operators and mixing of basic and probabilistic formulas are not allowed. Hence, expressions $x : \sigma \vee P_{\geq t}y : \sigma \to \tau$ and $P_{\geq s}P_{\geq t}M : \tau$ are not well defined formulas of the logic $\mathsf{P}\Lambda_{\to}$. Since we did not want the language to contain higher-order probabilistic formulas, we defined it by layering formulas into basic and probabilistic. Some examples of well defined formulas in this language are: $x : \sigma \to \tau \wedge y : \sigma$,
$x : \sigma \to \tau \wedge y : \sigma \Rightarrow xy : \tau$, $P_{=1}(x : \sigma \to \tau \wedge y : \sigma) \Rightarrow P_{=1}xy : \tau$.

A *semantics* of $\mathsf{P}\Lambda_{\rightarrow}$, we have proposed, is a Kripke-style semantics based on the possible-world approach, where each possible world is a lambda model. The crucial part in the proof of strong completeness of $\mathsf{P}\Lambda_{\rightarrow}$ is the fact that the simple type assignment of [4] is sound and complete with respect to the models, which are possible worlds in this structure. Hence, a type assignment, which is sound and complete was necessary. A probability measure is defined on the algebra of subsets of W, where W is the set of possible worlds. We have presented an infinitary axiomatization of $\mathsf{P}\Lambda_{\rightarrow}$, which consists of: (1) axioms for the classical propositional logic, (2) axioms for probabilistic logics and (3) two groups of inference rules. Rules from the first group can be applied only to lambda statements and those rules define simple type assignment. In the second group, we have three rules: the first one can be applied to both basic and probabilistic formulas (modus ponens), the second one can be applied to basic formulas and the third one can be applied only to probabilistic formulas.

The main results are the corresponding soundness and strong completeness of $\mathsf{P}\Lambda_{\rightarrow}$. The proof of the strong completeness, as well as construction of the canonical model, relies on two key facts. The first one is the completeness of the simple type assignment with respect to the simple semantics, proved in [4], and the second one is the existence of a maximal consistent extension of a consistent set.

A framework for probabilistic reasoning about typed terms is also provided in [1]. The authors proposed a probabilistic type theory in order to formalize computation with statements of the form "a given type is assigned to a given situation with probability p", so the approach is similar to ours. However, the difference is that neither soundness nor completeness issues were discussed. The developed theory was used for analyzing semantic learning of natural languages in the domain of computational linguistics.

Currently, we are extending the results and the techniques developed in [3] to lambda calculus with intersection types, as discussed in [2], exploiting the soundness and completeness of intersection type assignment with respect to the filter lambda models. The axiomatization in [3] includes classical propositional logic, because we wanted to reason about probabilities in a classical way, but one of the ideas for future work is the axiomatization that includes intuitionistic propositional logic.

References

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

[2] Silvia Ghilezan, Jelena Ivetić, Zoran Ognjanović, and Nenad Savić. Towards probabilistic reasoning about lambda terms with intersection types. In *TYPES 2016 - The 22nd Conference on Types for Proofs and Programs*, pages 59–60, 2016.

[3] Silvia Ghilezan, Jelena Ivetić, Simona Kašterović, Zoran Ognjanović, and Nenad Savić. Probabilistic reasoning about simply typed lambda terms. In *Logical Foundations of Computer Science - LFCS 2018*, volume 10703 of *Lecture Notes in Computer Science*, pages 170–189, 2018.

[4] J. Roger Hindley. The completeness theorem for typing lambda-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[5] Nebojša Ikodinović, Zoran Ognjanović, Miodrag Rašković, and Zoran Marković. First-order probabilistic logics and their applications. *Zbornik radova, subseries Logic in Computer Science, Matematički institut*, 18(26):37–78, 2015.

[6] Z. Ognjanović, M. Rašković, and Z. Marković. *Probability Logics: Probability-Based Formalization of Uncertain Reasoning*. Springer, 2016.

Towards a mechanized soundness proof for sharing effects

Paola Giannini[1], Tim Richter[2], Marco Servetto[3], and Elena Zucca[4]

[1] Università del Piemonte Orientale, Italy giannini@di.unipmn.it
[2] Universität Potsdam, Germany tim@cs.uni-potsdam.de
[3] Victoria University of Wellington, New Zealand marco.servetto@ecs.vuw.ac.nz
[4] Università di Genova, Italy elena.zucca@unige.it

Summary In previous work [4, 3], we have introduced an imperative calculus equipped with a type and effect system which infers *sharing* possibly introduced by the evaluation of an expression. The calculus is *pure* in the sense that reduction is defined on language terms only, since they directly encode store. The advantage of this non-standard operational model with respect to a conventional model using a global memory is that reachability relations among references are partly encoded by scoping. Moreover, sharing is represented as an equivalence relation among variables. We have implemented in Coq the type and effect system and (partly) the operational semantics. We plan to complete the implementation, as detailed below, and to mechanize the proof of soundness. We argue that the Coq implementation nicely illustrates the advantages of our purely syntactic model, since proofs can be carried out inductively, without requiring more complicated techniques such as, e.g., bisimulation.

Syntax and operational semantics The syntax of expressions is given by:

$e ::= x \mid e.f \mid e.m(e_1, \ldots, e_n) \mid e.f{=}e' \mid \texttt{new}\ C(e_1, \ldots, e_n) \mid \{d_1 \ldots d_n\ e\}$ expression
$d ::= T\ x{=}e;$ declaration

We assume a Featherweight Java style class table where method bodies are expressions from this grammar. An expression can be a variable, a field access, a method invocation, a field assignment, a constructor invocation and a block consisting of a sequence of declarations and a body. A declaration specifies a type, a variable and an initialization expression.

The following is an example of reduction sequence in the calculus. We emphasize at each step the declarations which can be seen as the store (in grey) and the redex which is reduced (in a box). We feel free to also use expressions of primitive types such as `int`, we omit the brackets of the outermost block, and abbreviate $\{T\ x{=}e;\ e'\}$ by $e;e'$ when x does not occur free in e'.

```
D z=new D(0); C x=new C(z,z); [C y=x;] D w=new D(y.f1.f+1); x.f2=w; x ⟶
D z=new D(0); C x=new C(z,z); D w=new D([x.f1].f+1); x.f2=w; x ⟶
D z=new D(0); C x=new C(z,z); D w=new D([z.f]+1); x.f2=w; x ⟶
D z=new D(0); C x=new C(z,z); D w=new D([0+1]); x.f2=w; x ⟶
D z=new D(0); C x=new C(z,z); D w=new D(1); [x.f2=w]; x ⟶
D z=new D(0); C x=new C(z,w); D w=new D(1); x
```

The main idea is to use local variable declarations to directly represent the store. That is, a declared variable is not replaced by its value, as in standard `let`, but the association is kept and used when necessary.

Assuming a program (class table) where class `C` has two fields `f1` and `f2` of type `D`, and class `D` has an integer field `f`, in the initial term the first two declarations can be seen as a store which associates to `z` an object of class `D` whose field contains `0`, and to `x` an object of class `C` whose two fields contain (a reference to) the previous object. The first reduction step eliminates an alias, by replacing occurrences of `y` by `x`. The next three reduction steps compute `x.f1.f+1`, by performing two field accesses and one sum. The last step performs a field assignment. The final result of the evaluation is an object of class `C` whose fields contain two objects of class `D`, whose field contains `0` and `1`, respectively.

Since our calculus smoothly integrates memory representation with shadowing and α-conversion, reading (or, symmetrically, updating) a field could cause scope extrusion. To avoid this problem, such reduction steps are only allowed modulo a *congruence* relation $\cong$, which captures structural equivalence, as in π-calculus [6]. To ensure that, in such a congruence, declarations are moved from a block to the directly enclosing block only when this is safe, blocks are annotated, during typechecking, with local variables which will be connected to the result of the block.

Type system Given an expression e, the type system computes a *sharing relation* $\mathcal{S}$ which is an equivalence relation on a set containing e's free variables and an additional distinguished variable `res` denoting the result of e. The fact that two variables, say x and y, are in the same equivalence class in $\mathcal{S}$, means that the evaluation of e can possibly introduce sharing between x and y, that is, connect their reachable object graphs, so that a modification of (a subobject of) x could affect y as well, or conversely. For instance, let e_b be the expression $x.f$=y;$z.f$, the sharing effects computed by the type system are two equivalence classes: $\{x, y\}$ and $\{\texttt{res}, z\}$. The typing judgment has shape $\Gamma \vdash e : T \,|\, \mathcal{S} \rightsquigarrow e'$ where Γ is an assignment of types to variables, $\mathcal{S}$ is a sharing relation on $\texttt{dom}(\Gamma) \cup \{\texttt{res}\}$ and e' is an *annotated expression*, where blocks are annotated by the local variables which will be (possibly) connected with the result of the body. For instance, if class C has a field of class D, we have that $x : C, z : C, y : D \vdash e_b : D \,|\, \{x, y\}\{\texttt{res}, z\} \rightsquigarrow e_b$. Moreover, let d be $C\ x$=new C(new D()); $C\ z$=new C(new D()); $D\ y$=new D(); we have that $\vdash \{d\ \ e_b\} : D \,|\, \epsilon \rightsquigarrow \{^{\{z\}} d\ \ e_b\}$ where ϵ is the identity equivalence relation (in this case empty since there are no free variables).
This type and effect system which *infers* sharing effects is very expressive, notably it detects *uniqueness* of references in much more situations than others based on *recovery* [5, 1].

Related work Several mechanized proofs of soundness in Coq have been presented in the literature, even for sophisticated type systems. However, we are not aware of such proofs for type systems ensuring uniqueness in a highly expressive way. Notably, the soundness of the language presented in [5] is done by embedding the types denotation into a sound program logic [2] and not formalizing the language directly.

Towards a mechanized proof We have implemented in Coq the type and effect system and the reduction rules. This required the formalization of sharing relations, which are built on top of equivalence relations on finite sets. We explored different ways to formalize the latter, converging on a definition as an inductive family, properties of which we hope to prove concisely using Coq's new dependent pattern matching capabilities [7]. The current code can be found at `//github.com/paola-giannini/sharing`. To complete the implementation of the operational semantics, we need an oriented version of the congruence relation on terms to be applied before reduction steps, analogously to α-conversion. Then, we plan to mechanize the full proof of soundness. We argue that the Coq implementation nicely illustrates the advantages of our purely syntactic model, since proofs can be carried out inductively, without requiring more complicated techniques such as, e.g., bisimulation. To support this claim we plan to compare our formalization with the one of [5].

References

[1] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela, editors, *International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015*, pages 1–12. ACM Press, 2015.

[2] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 287–300. ACM, 2013.

[3] Paola Giannini, Marco Servetto, and Elena Zucca. Tracing sharing in an imperative pure calculus: extended abstract. In *FTfJP'17 - Formal Techniques for Java-like Programs*, pages 6:1–6:6. ACM Press, 2017.

[4] Paola Giannini, Marco Servetto, and Elena Zucca. A type and effect system for sharing. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *OOPS'17 - Object-Oriented Programming Languages and Systems, Track at SAC'17 - ACM Symp. on Applied Computing*, pages 1513–1515. ACM Press, 2017.

[5] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In Gary T. Leavens and Matthew B. Dwyer, editors, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012)*, pages 21–40. ACM Press, 2012.

[6] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[7] Matthieu Sozeau and Cyprien Mangin. Equations - a function definition plugin. https://github.com/mattam82/Coq-Equations.

Syntactic investigations into cubical type theory

Hugo Herbelin
INRIA - IRIF - University Paris Diderot

Abstract

This is a proposal for a talk exploring syntactic variations of cubical type theory, with the following motivations in mind:

- providing a syntax of cubical type theory close to the syntax of (some semi-strict variant of) an ω-groupoid,
- seeing equivalence as the definition of equality on types, hence with univalence satisfied by construction, as already explored e.g. by Altenkirch and Kaposi [AK18], or Polonsky [Pol15].

Additionally, we shall develop a set of notations to reason about (hyper)cubes in arbitrary dimensions, roughly in the continuation of Licata and Brunerie [LB15, Bru16], using nesting of the heterogeneous formulation of equality characterized by abstraction over an interval.

In traditional Martin-Löf's type theory, the tower of iteration of the identity type over a type has a structure of globular set which extends to a structure of weak ω-groupoid [LeF08, vG08], or actually even of a semi-strict ω-groupoid since one of the laws of neutrality of reflexivity holds definitionally.

If one moves to cubical type theory [CCHM16, ABC$^+$17], the same can be said with the difference that the structure of equality given by dependent[1] abstraction and application over a formal interval is the one of a (Cartesian) symmetric cubical set (without connections nor inverses at this stage).

In the first part of the talk, we shall define a couple of notations for reasoning about (hyper)cubes in all dimensions, emphasizing that the laws of a (Cartesian) symmetric cubical sets hold definitionally in cubical type theory (see also [vL18] for an exploration of the properties of equality seen as dependent product from an interval).

In the second part of the talk, we shall consider a variant of cubical type theory with connections and inverses but also with ordinary composition rather than "Kan composition" along a tube of faces. Having composition, connections and inverses primitive makes the syntax of type theory explicitly closer to the one of cubical ω-categories and cubical ω-groupoids, but with a semi-strict choice of rules (note: our motivation is pragmatical and we have no claim that this particular choice of semi-strictness is adequate for synthetic homotopy).

Our interval has no structure, besides supporting variables of interval. The rules for equality shall be the following ones:

$$\frac{\Gamma \vdash \epsilon : A =_\theta B \quad \Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash t =_\epsilon u : \mathsf{U}} \qquad \frac{\Gamma, i \vdash t : A}{\Gamma \vdash \lambda i.t : t[0/i] =_{\lambda i.A} t[1/i]} \qquad \frac{\Gamma \vdash v : t =_\epsilon u \quad i \in \Gamma}{\Gamma \vdash vi : \epsilon i}$$

where, as an experiment, we take substitution of an interval variable by its endpoints as an effective (i.e. meta-level) operation.

The typing rules for composition, inverse and connections are the following:

$$\frac{\Gamma \vdash p : t =_\epsilon u \qquad \Gamma \vdash q : u =_\zeta v}{\Gamma \vdash q \circ p : t =_{\zeta \circ \epsilon} v} \qquad \frac{\Gamma \vdash p : t =_\epsilon u}{\Gamma \vdash p^{-1} : u =_{\epsilon^{-1}} t}$$

[1] as in [CCHM16, Section 9] or [ABC$^+$17]

$$\frac{\Gamma \vdash \epsilon : t =_\theta u}{\Gamma \vdash \ulcorner \epsilon : \epsilon =_{\epsilon \widetilde{=}_{\ulcorner \theta} \widehat{u}} \widehat{u}} \qquad \frac{\Gamma \vdash \epsilon : t =_\theta u}{\Gamma \vdash \epsilon_\lrcorner : \widehat{t} =_{\widehat{t} \widetilde{=}_{\theta_\lrcorner} \epsilon} \epsilon}$$

where $t \widetilde{=}_\epsilon u$ abbreviates $\lambda i.(t\, i =_{\epsilon i} u\, i)$ and $\widehat{t}$ abbreviates $\lambda i.t$ for i not occurring in t (i.e. reflexivity).

Up to now, the framework only gives a structure to equality. It remains to define equality for each type constructor. For the universe, equality is taken to be equivalence:

$$\frac{\begin{array}{ll}\Gamma \vdash A : \mathsf{U} & \Gamma \vdash B : \mathsf{U} \\ \Gamma, a : A \vdash f(a) : B & \Gamma, b : B \vdash g(b) : A \\ \Gamma, a : A \vdash p(a) : a =_{\widehat{A}} g(f(a)) & \Gamma, b : A \vdash q(b) : f(g(b)) =_{\widehat{B}} b\end{array}}{\Gamma \vdash \{f(a); g(b); p(a); q(b)\}_{a:A,b:B} : A =_{\widehat{\mathsf{U}}} B}$$

with the equivalence internally turned into an adjoint one. We can then show that equivalences can be equipped with the structure given by composition, connections, inverses and abstraction over a formal interval, so that the generic properties of equality apply. We then get type coercions for free (and thus transport) as the projections of a proof of equality of types.

We shall incidentally give a proof of the following dependent form of functional extensionality by allowing term variables to be instantiated by interval variables:

$$\frac{\begin{array}{ll}\Gamma, i \vdash A : \mathsf{U} & \Gamma, i, a : A \vdash B : \mathsf{U} \\ \Gamma \vdash f : \Pi a{:}A[0/i].\, B[0/i] & \Gamma \vdash g : \Pi a{:}A[1/i].\, B[1/i] \\ p : \Pi a_0{:}A[0/i].\, \Pi a_1{:}A[1/i].\, \Pi e{:}a_0 =_{\lambda i.A}\, a_1.\, f\, a_0 =_{\lambda i.B[e\,i/a]}\, g\, a_1 & \end{array}}{\Gamma \vdash \mathsf{fundepext}(p) : f =_{\lambda i.\Pi a A.\, B}\, g}$$

References

[ABC+17] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. Cartesian cubical type theory. 2017.

[AK18] Thorsten Altenkirch and Ambrus Kaposi. Towards a Cubical Type Theory without an Interval. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *LIPIcs*, pages 3:1–3:27. Schloss Dagstuhl, 2018.

[Bru16] Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory*. PhD thesis, June 2016. Available at https://arxiv.org/pdf/1606.05916.pdf.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[LB15] Daniel R. Licata and Guillaume Brunerie. A cubical approach to synthetic homotopy theory. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 92–103. IEEE Computer Society, 2015.

[LeF08] Peter LeFanu Lumsdaine. Weak omega-categories from intensional type theory. *ArXiv e-prints*, December 2008.

[Pol15] Andrew Polonsky. Extensionality of lambda-*. In Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau, editors, *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *LIPIcs*, pages 221–250. Schloss Dagstuhl, 2015.

[vG08] Benno van den Berg and Richard Garner. Types are weak omega-groupoids. *ArXiv e-prints*, December 2008.

[vL18] Twan van Laarhoven. Type theory with indexed equality - the theory. https://twanvl.nl/blog/hott/ttie-theory, 2018.

Beyond the limits of the Curry-Howard isomorphism

Reinhard Kahle[1] and Anton Setzer[2]

[1] CMA and DM, FCT, Universidade Nova de Lisboa, Portugal. kahle@mat.uc.pt
[2] Dept. of Computer Science, Swansea University. a.g.setzer@swan.ac.uk

The well-known Curry-Howard isomorphism relates functions with proofs and can be considered as one of the conceptional bases of Martin-Löf's type theory (MLTT).

For our considerations, the crucial correspondence is the one between (intuitionistic) proofs of an implication $A \to B$ and functions of the type $A \to B$. To make sense out of this correspondence, the functions need to be, of course, *total*, i.e., for every element of A the function needs to associate an element of B.

Although totality of functions is a desirable property, it does not match with computational reality. It is not only the case that non-terminating computations appear natural. Even more importantly, there is no Turing complete computable model of computation in which all functions are total, therefore there is no Turing complete programming language based on total functions.

Partial functions are easily integrated in *type-free* contexts, and we may address shortly some interesting historical considerations regarding *functional self-application*, cf. [Kah07, Appendix]. The main aim of this talk is, however, to illustrate the role of partial functions in the formalization of the *extended predicative Mahlo universe* (EPM), [KS10].

A weakly Mahlo cardinal is a regular cardinal κ such that for every function $f : \kappa \to \kappa$ there exists a regular cardinal $\pi < \kappa$ s.t. $f : \pi \to \pi$ [Rat90]. This definition has been translated into MLTT [Set00] and Feferman's theory of Explicit Mathematics (EM) [JS01]. In EM, a Mahlo universe is a universe M such that for every $a \in \mathsf{M}$ and $f : \mathsf{M} \to \mathsf{M}$ there exists a subuniverse $\mathsf{m}(a, f)$ of M, which is an element of M, contains a, and is closed under f.

This axiomatization, which we call axiomatic Mahlo (AxM), is clearly highly impredicative, since, if viewed as an introduction rule, the definition of M requires to add $\mathsf{m}(a, f)$ for all total $f : \mathsf{M} \to \mathsf{M}$, where the set of total f is only known after M is complete.

In the EPM, formulated in EM, the totality of $f : \mathsf{M} \to \mathsf{M}$ is no longer required. Instead one requires f only to be total on the subset $\mathsf{m}(a, f)$ of M, and not on elements added after the addition of $\mathsf{m}(a, f)$ to M – the reason for adding $\mathsf{m}(a, f)$ is not destroyed by the addition of $\mathsf{m}(a, f)$ or any element added after $\mathsf{m}(a, f)$. More precisely one tries to build subsets $\mathsf{m}(a, f)$ closed under f and a. If that succeeds $\mathsf{m}(a, f)$ is added to M. M is constructed from below, because the reason for adding $\mathsf{m}(a, f)$ depends only on $\mathsf{m}(a, f)$ and not on all of M.

It turns out that the permission of partial functions is crucial for the axiomatization of the EPM, and because of this, we have not been able to port it to MLTT yet. Due to the use of partial functions one can define in EPM an elimination rule for M, which makes M a *least Mahlo universe*. This implies that M contains only elements introduced by its introduction rules. In AxM such an elimination rule results in a contradiction (see [Pal98] for a proof in the context of MLTT).

The reason for calling the resulting theory *extended* predicative is that it goes beyond the proof theoretic definition of predicativity (i.e., theories with the proof-theoretic strength of Γ_0). It goes as well beyond its use in MLTT, where it seems to be limited to types defined by function types and inductive and inductive-recursive definitions [Dyb00, DS03]. In this sense the EPM goes beyond what is undoubtedly considered as predicative in MLTT.[1]

[1] One should note that many type theoretists consider the Mahlo universe in MLTT, which goes beyond inductive and inductive-recursive definitions, as predicative as well.

This result suggests, on the proof-theoretic side, that the formulation of a least Mahlo universe, and as well of a Mahlo universe which is without any doubts accepted as predicative in nature, may exceed the limit of MLTT (or at least require a paradigm shift in MLTT). On the more conceptional side, it challenges the restriction to total functions in the Curry-Howard correspondence, although the question, what the meaning of a "partial implication" could be, remains a desideratum (see also [Kah1x]).

References

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

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525 – 549, June 2000.

[JS01] Gerhard Jäger and Thomas Strahm. Upper bounds for metapredicative Mahlo in explicit mathematics and admissible set theory. *Journal of Symbolic Logic*, 66(2):935–958, 2001.

[Kah07] Reinhard Kahle. *The applicative realm*, volume 40 of *Textos de Matemática*. Departamento de Matemática, Universidade de Coimbra, 2007. Habilitation thesis, Fakultät für Informations- und Kognitionswissenschaften, Universität Tübingen.

[Kah1x] Reinhard Kahle. Is there a "Hilbert Thesis"? *Studia Logica*, To appear. Special Issue on General Proof Theory. Thomas Piecha and Peter Schroeder-Heister (Guest Editors).

[KS10] Reinhard Kahle and Anton Setzer. An extended predicative definition of the Mahlo universe. In Ralf Schindler, editor, *Ways of Proof Theory*, Ontos Series in Mathematical Logic, pages 309 – 334. Ontos Verlag, 2010.

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

[Rat90] Michael Rathjen. Ordinal notations based on a weakly Mahlo cardinal. *Archive for Mathematical Logic*, 29:249 – 263, 1990.

[Set00] Anton Setzer. Extending Martin-Löf type theory by one Mahlo-universe. *Arch. Math. Log.*, 39:155 – 181, 2000.

Closure Conversion for Dependent Type Theory, With Type-Passing Polymorphism*

András Kovács

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

Closure conversion is an early translation step in the compilation of functional languages, which converts functions with potential free variable occurrences to pairs consisting of environments and closed functions. Minamide et al. [2] described type-preserving closure conversion for a polymorphic language. They considered an *intensional* or *type-passing* implementation of polymorphism, which enables different memory layouts for differently typed runtime objects, and necessitates that runtime type representations are passed to polymorphic functions. In contrast, *type-erasing* polymorphism (as in [3]) removes types during compilation, mandating uniform runtime representations (although with potential layout-changing optimizations, such as unboxing).

Generalizing type-passing polymorphism to dependent type theories would allow precise specification of memory layout using dependent types. For example, Σ-types may represent two values next to each other in memory, where the size and layout of the second field depends on the value of the first field. Hence, runtime objects would be described by type-theoretic universes instead of simple statically known layout schemes. Also, a closure-converted type theory with precise control over memory layout could be useful as an intermediate language even if types are erased somewhere on the way to machine code.

The current work is a first step in this direction. I describe a dependent type theory with a predicative universe hierarchy, Σ-types, Π-types with *closed* inhabitants and primitive closure objects. Also, types and runtime type codes are distinguished by Tarski-style universes, and type codes are themselves closure converted. Consistency for this theory is proved with a standard type-theoretic model. Then, it is proved that the general function space with term formation in non-empty contexts is admissible in this theory. General functions are represented as closures and term formation corresponds to closure building. The expected β and η rules also hold for this function space. Then, a closure conversion translation into this theory is presented, from a source theory with predicative universes and dependent functions. Injectivity, preservation of typing and preservation of conversion are proven for the translation.

Closures and type codes in the target theory

The target theory has predicative universes U_i with decoding El, Σ-types, closed function types (denoted $(a : A) \to B$) and closure types $\mathsf{Cl}\,(a : A)\,B$. Closed functions differ from usual functions only in the term formation rule: λ-abstraction is only valid in the empty context (denoted $\cdot$). For closures, there are rules for type and term formation, elimination, and η and β conversion, presented in this order:

$$\frac{\Gamma \vdash A\,\mathsf{type}_i \quad \Gamma, a : A \vdash B\,\mathsf{type}_j}{\Gamma \vdash \mathsf{Cl}\,(a : A)\,B\;\mathsf{type}_{\max(i,j)}} \qquad \frac{\cdot \vdash E : \mathsf{U}_i \quad \Gamma \vdash \mathit{env} : \mathsf{El}\,E \quad \cdot \vdash t : (\mathit{ea} : \Sigma(e : \mathsf{El}\,E).A) \to B}{\Gamma \vdash \mathsf{pack}\,E\,\mathit{env}\;t : \mathsf{Cl}\,(a : A[e \mapsto \mathit{env}])\,(B[\mathit{ea} \mapsto (\mathit{env}, a)])}$$

*This work was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

$$\frac{\Gamma \vdash t : \mathsf{Cl}\,(a : A)\,B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B[a \mapsto u]} \qquad \frac{\Gamma \vdash t : \mathsf{Cl}\,(a : A)\,B \quad \Gamma \vdash u : \mathsf{Cl}\,(a : A)\,B \quad \Gamma,\, a : A \vdash t\,a \equiv u\,a}{\Gamma \vdash t \equiv u}$$

$$(\mathsf{pack}\,E\;env\;t)\,u \equiv t\,(env,\,u)$$

The type code inhabitants of U_i may also contain closures. This is required for efficient runtime computation of type dependencies in a potential type-passing implementation. Decoding with El computes types from codes by applying closures as needed. Rules for Cl codes are listed below; cases for other types are analogous.

$$\frac{\Gamma \vdash A : \mathsf{U}_i \quad \Gamma \vdash B : \mathsf{Cl}\,(\mathsf{El}\,A)\,(\mathsf{U}_j)}{\Gamma \vdash \mathsf{Cl}'\,A\,B : \mathsf{U}_{\max(i,j)}} \qquad \mathsf{El}\,(\mathsf{Cl}'\,A\,B) \equiv \mathsf{Cl}\,(a : \mathsf{El}\,A)\,(\mathsf{El}\,(B\,a))$$

We use an abstract closure representation, in contrast to [2], where closures are derived from existential and translucent types. This is because of the need to capture environments at arbitrary universe levels, which precludes Σ representations.

Unknown to the author at the time of submission, Ahmed and Bowman [1] developed closure conversion for the Calculus of Constructions, and used a similar closure representation for the same reasons, with analogous β and η rules. The main difference to the current work is that they don't consider closure conversion for type codes, only for terms. There are also a number of technical differences, for instance, the current work uses a predicative hierarchy instead of two universes with an impredicative base universe, and does not consider deterministic reduction, only a non-directed conversion relation.

Admissibility of general function space

The main goal is to build a term of $\mathsf{Cl}\,(a : A)\,B$ from some $\Gamma, a : A \vdash t : B$, in a way such that β, η and substitution rules hold. Closure building is defined mutually with quoting operations on well-formed contexts and types:

- From each Γ, we construct a closed code $\mathsf{quote}\;\Gamma$ for the corresponding iterated Σ-type, along with an isomorphism between Γ and the singleton context containing $\mathsf{El}\,(\mathsf{quote}\;\Gamma)$, consisting of two back-and-forth substitutions.
- From each $\Gamma \vdash A\;\mathsf{type}_i$, we construct $\Gamma \vdash \mathsf{quote}\,A : \mathsf{U}_i$, such that El retracts quote, and quote is natural with respect to type substitution. Quoting to type codes here involves building closures which compute type dependencies, as we have seen for the Cl example.
- Closures are built by pack-ing together quote-ed environment types, environments (given from $\Gamma \to \mathsf{El}\,(\mathsf{quote}\;\Gamma)$ substitutions) and closed function bodies (given by closing the t input function bodies using the $\mathsf{El}\,(\mathsf{quote}\;\Gamma) \to \Gamma$ substitutions).

References

[1] William J Bowman and Amal Ahmed. Typed closure conversion for the calculus of constructions. 2018.

[2] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–283. ACM, 1996.

[3] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999.

On the Role of Semisimplicial Types

Nicolai Kraus

University of Nottingham

Abstract

Constructing semisimplicial types is a well-known open problem in homotopy type theory. I explain why I believe that this problem is highly important. This talk proposal is based on several papers that are already available as well as on work in progress.

What are semisimplicial types? Let us write $\mathcal{U}$ for a universe in MLTT/HoTT. A *semisimplicial type* restricted to level 2 is a tuple (A_0, A_1, A_2) of the following types, where uncurrying is done implicitly: $A_0 : \mathcal{U}$

$$\begin{aligned} &A_1 : A_0 \to A_0 \to \mathcal{U} \\ &A_2 : (x, y, z : A_0) \to A_1(x, y) \to A_1(y, z) \to A_1(x, z) \to \mathcal{U} \end{aligned} \tag{1}$$

We can interpret A_0 as a type of points, A_1 as a type of (directed) lines between two given points, and A_3 as a type of "triangle fillers". On the next level, we would add a type family A_3 indexed over four points, six lines, and four triangle fillers forming a tetrahedron, and so on.[1]

Can we define semisimplicial types in HoTT? It is unknown whether there is a type family $F : \mathbb{N} \to \mathcal{U}_1$ such that $F(n)$ encodes the type of tuples $(A_0, \dots, A_n)$ in any suitable way in "book HoTT" (the type theory developed in [14]). This is a major open problem in homotopy type theory, known as the problem of **defining semisimplicial types**.

The problem has been considered so significant that other type theories which allow solutions have been suggested. The first is Voevodsky's *homotopy type system (HTS)* which enables us to reason about strict equality. The *two-level type theories (2LTTs)* as presented by Altenkirch, Annenkov, Capriotti and myself [1, 4] are variants of HTS which offer some choices. A form of 2LTT has been used by Boulier and Tabareau to define a model structure on the universe [5]. Another alternative to HTS is the *logic-enriched type theory* by Part and Lou [12]. As far as I know, a definition of semisimplicial types is also possible in the *computational higher type theory* of Angiuli, Favonia, Harper, and Wilson (see [3] and related papers).

Why is this problem interesting? Since (homotopy) type theory is a "theory based on ∞-groupoids", it is natural to attempt the development of a theory of higher categories, but it is open how to even define the notion of an $(\infty, 1)$-category in "book HoTT". In a setting with semisimplicial types, Capriotti and I have suggested and studied *complete semi-Segal types* which so far work well [6]. Related is the work by Sattler and myself: we can define types of diagrams over many different index categories, and we can show that these definitions are well-behaved [11].

A further important question is whether homotopy type theory can serve as its own metatheory, which one would expect from a foundation of mathematics. If this is the case, then semisimplicial types can be constructed [13]. I have hope that the opposite direction can be shown as well; this would require a version of Altenkirch and Kaposi's "type theory in type theory" [2] without set-truncation.

[1] Remark: This is an approach to encode type-valued presheaves over the category Δ_+ (the category of finite nonzero ordinals and strictly increasing functions) without having to talk about functor laws; it is inspired by the Reedy model structure for functor categories.

Do semisimplicial types also matter for synthetic homotopy theory? In the very impressive existing work in synthetic homotopy theory (e.g. the results of the HoTT book [14]), it has not been necessary to consider infinite coherence structures explicitly because clever encodings of the relevant data have been used to avoid such higher structures. One example for this is the the notion of a *half-adjoint equivalence*, where one gives only one out of two equations on level 2; the slightly less clever way would be to use both equations on level 2, then two coherence equations on level 3, and so on, already generation an infinite (but in this case less complicated) tower. It is not clear that such encodings are possible in all situations that might turn up in HoTT. One example that I would like to discuss in the talk is the open problem whether the suspension of a set is always 1-truncated [14, Exercise 8.2]. A similar question which I have posted on the HoTT mailing list [9] asks whether "adding a path" to a 1-type (or higher) preserves its truncation level. I conjecture that both can be answered positively if semisimplicial types are available in the type theory (see [10] for a partial result); one way to see a connection is applying the encode-decode method [14, Chp 2.12] which in the case of these problems seems to require expressing coherence towers. One of the arguments I am using to attack the problem is a special case of another of my results that depend on semisimplicial types, namely the fact that functions $\|A\|_{-1} \to B$ correspond to *coherently constant functions* $A \to B$ [7, 8].

References

[1] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending homotopy type theory with strict equality. *CSL'16*, 2016. http://dx.doi.org/10.4230/LIPIcs.CSL.2016.21.

[2] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *POPL'16*, 2016. https://doi.org/10.1145/2837614.2837638.

[3] Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher-dimensional type theory. *POPL'17*, 2017. https://doi.org/10.1145/3009837.3009861.

[4] Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. Two-level type theory and applications. 2017. arXiv:1705.03307.

[5] Simon Boulier and Nicolas Tabareau. Model structure on the universe in a two level type theory. 2017. https://hal.archives-ouvertes.fr/hal-01579822.

[6] Paolo Capriotti and Nicolai Kraus. Univalent higher categories via complete semi-Segal types. *POPL'18*, 2018. http://doi.acm.org/10.1145/3158132.

[7] Nicolai Kraus. The general universal property of the propositional truncation. *TYPES'14*, 2015. http://dx.doi.org/10.4230/LIPIcs.TYPES.2014.111.

[8] Nicolai Kraus. *Truncation Levels in Homotopy Type Theory*. PhD thesis, School of Computer Science, University of Nottingham, 2015.

[9] Nicolai Kraus. Does "adding a path" preserve truncation levels?, 2018. On the HoTT mailing list, https://groups.google.com/d/msg/homotopytypetheory/gVmcvaOeD5c/IfCwA1lcAwAJ.

[10] Nicolai Kraus and Thorsten Altenkirch. Free higher groups in homotopy type theory. *LiCS'18*, 2018. https://doi.org/10.1145/3209108.3209183.

[11] Nicolai Kraus and Christian Sattler. Space-valued diagrams, type-theoretically (extended abstract). 2017. arXiv:1704.04543.

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

[13] Michael Shulman. Homotopy type theory should eat itself (but so far, it's too big to swallow), 2014. Blog post, homotopytypetheory.org/2014/03/03/hott-should-eat-itself.

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

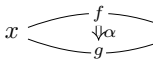
Types are weak omega-groupoids, in Coq

Ambroise Lafont[1], Tom Hirschowitz[2], and Nicolas Tabareau[3]

[1] IMT Atlantique, Nantes, France
[2] CNRS, Université Savoie Mont-Blanc, France
[3] Inria, Nantes, France

The problem: types and weak ω-groupoids, internally One of the discoveries underlying the recent homotopical interpretation of Martin-Löf type theory is the fact that, for any type, the tower of its iterated Martin-Löf *identity* types gives rise to a weak ω-groupoid [LL10, vdBG11]. Some motivation has recently been put forward [ALR14, HT15] for performing this construction *internally*. However, first investigations have stumbled upon the following tension: (1) on the one hand, in order to semantically agree with the standard notion, ω-groupoids should be based on *sets*, i.e., 'discrete' types; (2) on the other hand, constructing the weak ω-groupoid associated to any non-discrete type appears to require basing them on general types.

The latter constraint is clear, but let us give a bit more detail about the former. As previous authors, we adopt Brunerie's definition of weak ω-groupoids [Bru16], which seems most directly amenable to internalization. Roughly, a weak ω-groupoid is a model (in types) of a specific, very simple type theory. In the most basic version of this theory, contexts correspond to finite globular sets, types to pairs of parallel cells, and terms to cells. Thus, for example, the globular set below left is modeled by the context below right



$$x : \star, y : \star, f : x =_\star y, g : x =_\star y, \alpha : f =_{x=_\star y} g \vdash .$$

Example types in it are $f =_{x=_\star y} g$ and $\alpha =_{f=_{x=_\star y} g} \alpha$; and an example term of the former type is α itself. In this basic version, models of the theory are merely globular types. The idea is then to enrich the type theory with term formers corresponding to composition, identities, associativity, and the whole standard package of higher coherences. This is done by first identifying a class of *contractible* contexts which correspond to pasting schemes. E.g., pasting schemes for binary composition and identities for 1-cells are given by

$$x : \star, y : \star, f : x =_\star y, z : \star, g : y =_\star z \vdash \qquad \text{and} \qquad x : \star \vdash .$$

Such contexts may be defined inductively, which yields a different judgement $\Gamma \vdash_c$. The crucial rule then (roughly) says that any type A in any contractible context Γ in inhabited by a term $\mathbf{coh}_{\Gamma,A}$, called a *coherence,* which amounts to saying that the corresponding pasting scheme admits a composite. E.g., composition is the coherence obtained for the context above left, with $A = (x =_\star z)$, and identity corresponds to the one above right with $A = (x =_\star x)$.

Now the difficulty evoked in item (1) above arises in the definition of models of this type theory, which is polluted with coherence conditions. Typically, a naïve approach could start by defining a model to consist of a type $[\![\Gamma]\!]$ for each context Γ, plus for each $\Gamma \vdash A$ a family $[\![A]\!]_\Gamma$ indexed by the elements of $[\![\Gamma]\!]$, and for each term $\Gamma \vdash M : A$ a section of this family. But for any $\Gamma \vdash B$, there is another family $[\![A]\!]_{\Gamma,B}$, and we certainly want any model to satisfy the condition that the latter is precisely given by $(\gamma, b) \mapsto [\![A]\!]_\Gamma(\gamma)$, for any $\gamma \in [\![\Gamma]\!]$ and $b \in [\![B]\!]_\Gamma(\gamma)$. Such constraints are hard to specify without any redundancy. They thus generate higher constraints, and so on, which quickly becomes intractable.

A possible solution [ALR14] consists in taking models in *sets,* i.e., types with discrete homotopy type, which considerably improves the situation but gives up item (2).

Two-level type theory We here avoid this dilemma, by working in a *2-level* type theory [ACK16], i.e., a type theory with two notions of equality, one *strict* and one *homotopical.* We formalize the construction in a simple 2-level extension of Coq [Laf]. The point is to use strict equality to axiomatize weak ω-groupoids: the constraints evoked above are required to hold strictly, which has the same taming effect as taking models in sets, while still accomodating models in general types. Homotopical equality may then be used to construct the weak ω-groupoid associated to a so-called *fibrant* type. This idea turned out to work, but only up to the following issues, which are arguably of lesser conceptual importance.

First, the construction of the weak ω-groupoid associated to a type is rejected by Coq's well-foundedness criterion. We thus consider a variant of Brunerie's type theory which only considers contractible contexts, and accordingly terms and types therein. This is enough for Coq to swallow the pill, but of course we should adapt our notion of model to compensate for the missing contexts. However, non-contractible contexts never yield new coherences, hence only have to do with globular structure, not weak ω-groupoid structure. We may thus define weak ω-groupoids as models of our restricted type theory *in globular types,* so that globular structure is built into them from the start.

A second issue is related to defining type theories internally. In [ALR14], the authors formalize Brunerie's type theory as an *intrinsic* syntax, i.e., as an inductive-inductive-recursive datatype following the typing rules directly, although they have to switch off Agda's termination checker. However, Coq does not support such definitions, so we follow the old school route: we define untyped syntax first and then typing judgements, and use the Uniqueness of Identity Proofs principle satisfied by strict equality to simulate the non-dependent inductive-inductive-recursive eliminator required to construct the weak ω-groupoid associated to a type.

Weak ω-categories Finally, slightly restricting contractible contexts and the coherence rule, we also formalize weak ω-categories of [FM17]. We leave the following consistency check for future work: does any Finster-Mimram ω-category with weakly invertible cells yield a Brunerie ω-groupoid?

References

[ACK16] T. Altenkirch, P. Capriotti, and N. Kraus, *Extending Homotopy Type Theory with Strict Equality*, CSL, LIPIcs, vol. 62, Schloss Dagstuhl, 2016.

[ALR14] T. Altenkirch, N. Li, and O. Rypáček, *Some constructions on ω-groupoids*, LFMTP, ACM, 2014.

[Bru16] G. Brunerie, *On the homotopy groups of spheres in homotopy type theory*, Thèse de doctorat, Université Nice Sophia Antipolis, June 2016.

[FM17] E. Finster and S. Mimram, *A type-theoretical definition of weak ω-categories*, LICS, IEEE, 2017.

[HT15] A. and T. Hirschowitz and N. Tabareau, *Wild omega-categories for the homotopy hypothesis in type theory*, TLCA (T. Altenkirch, ed.), LIPIcs, vol. 38, Schloss Dagstuhl, 2015.

[Laf] A. Lafont, *A Coq formalization of the proof that types are weak omega groupoids*, https://github.com/amblafont/weak-cat-type/tree/untyped2tt.

[LL10] P. LeFanu Lumsdaine, *Weak omega-categories from intensional type theory*, Logical Methods in Computer Science **6** (2010), no. 3.

[vdBG11] B. van den Berg and R. Garner, *Types are weak ω-groupoids*, Proc. of the London Mathematical Society **102** (2011), no. 2, 370–394.

Simulating Induction-Recursion for Partial Algorithms

Dominique Larchey-Wendling[1] and Jean-François Monin[2]

[1] Université de Lorraine, CNRS, LORIA
dominique.larchey-wendling@loria.fr
[2] Université Grenoble Alpes, CNRS, Grenoble INP, VERIMAG
jean-francois.monin@univ-grenoble-alpes.fr

Abstract

We describe a generic method to implement and extract partial recursive algorithms in Coq in a purely constructive way, using L. Paulson's `if-then-else` *normalization as a running example.*

Implementing complicated recursive schemes in a Type Theory such as Coq is a challenging task. A landmark result is the Bove&Capretta approach [BC05] based on accessibility predicates, and in case of nested recursion, simultaneous Inductive-Recursive (IR) definitions of the domain/function [Dyb00]. Limitations to this approach are discussed in e.g. [Set06, BKS16]. We claim that the use of (1) IR, which is still absent from Coq, and (2) an informative predicate (of sort `Set` or `Type`) for the domain, preventing its erasing at extraction time, can be circumvented through a suitable *bar inductive predicate*.

$$\begin{array}{l}\texttt{type}\ \Omega = \alpha \mid \omega\ \texttt{of}\ \Omega * \Omega * \Omega\\ \texttt{let rec nm}\ e = \texttt{match}\ e\ \texttt{with}\\ \quad \mid \alpha \Rightarrow \alpha\\ \quad \mid \omega(\alpha, y, z) \Rightarrow \omega(\alpha, \texttt{nm}\ y, \texttt{nm}\ z)\\ \quad \mid \omega(\omega(a,b,c), y, z) \Rightarrow \texttt{nm}(\omega(a, \texttt{nm}(\omega(b,y,z)), \texttt{nm}(\omega(c,y,z))))\end{array}$$

Figure 1: L. Paulson's `if-then-else` normalisation algorithm.

We illustrate our technique on L. Paulson's algorithm for `if-then-else` normalization [Gie97, BC05] displayed in Fig. 1. For concise statements, we use ω to denote the ternary constructor for `if`$\_$`then`$\_$`else`$\_$ expressions, and α as the nullary constructor for atoms. As witnessed in the third match rule $\omega(\omega(a,b,c),y,z)$, `nm` contains (two) nested recursive calls, making its termination depend on properties of its semantics. This circularity complicates the approach of well-founded recursion and may even render it unfeasible.

Our method allows to show these properties *after* the (partial) function `nm` is defined, as proposed in [Kra10], but without the use of Hilbert's ε-operator. We proceed purely constructively without any extension to the existing Coq system and the recursive definition of Fig. 1 can be *extracted as is* from the Coq term that implements `nm`.

We start with the inductive definition of the graph $\mathbb{G} : \Omega \to \Omega \to$ `Prop` of `nm` (Fig. 2) and we show its functionality.[1] Then we define the domain/termination predicate $\mathbb{D} : \Omega \to$ `Prop` as a bar inductive predicate with the three rules of Fig. 3.

$$\frac{}{\mathbb{G}\ \alpha\ \alpha} \qquad \frac{\mathbb{G}\ y\ n_y \qquad \mathbb{G}\ z\ n_z}{\mathbb{G}\ (\omega\ \alpha\ y\ z)\ (\omega\ \alpha\ n_y\ n_z)}$$

$$\frac{\mathbb{G}\ (\omega\ b\ y\ z)\ n_b \qquad \mathbb{G}\ (\omega\ c\ y\ z)\ n_c \qquad \mathbb{G}\ (\omega\ a\ n_b\ n_c)\ n_a}{\mathbb{G}\ (\omega\ (\omega\ a\ b\ c)\ y\ z)\ n_a}$$

Figure 2: Rules for the graph $\mathbb{G} : \Omega \to \Omega \to$ `Prop` of `nm`.

$$\frac{}{\mathbb{D}\ \alpha} \qquad \frac{\mathbb{D}\ y \qquad \mathbb{D}\ z}{\mathbb{D}\ (\omega\ \alpha\ y\ z)}$$

$$\frac{\mathbb{D}\ (\omega\ b\ y\ z) \qquad \mathbb{D}\ (\omega\ c\ y\ z) \qquad \forall n_b\ n_c, \mathbb{G}\ (\omega\ b\ y\ z)\ n_b \to \mathbb{G}\ (\omega\ c\ y\ z)\ n_c \to \mathbb{D}\ (\omega\ a\ n_b\ n_c)}{\mathbb{D}\ (\omega\ (\omega\ a\ b\ c)\ y\ z)}$$

Figure 3: Rules for the bar inductive definition of $\mathbb{D} : \Omega \to$ `Prop`.

There, we single out recursive calls using $\mathbb{G}$ but proceed by pattern-matching on e following the recursive scheme of `nm` of Fig. 1. Then we define `nm_rec` $: \forall e\ (D_e : \mathbb{D}\ e), \{n \mid \mathbb{G}\ e\ n\}$ as a fixpoint using D_e to ensure termination. However, the term `nm_rec` $e\ D_e$ does not use D_e to compute: the value n satisfying $\mathbb{G}\ e\ n$ is computed by pattern-matching on e and recursion, following the scheme of Fig. 1.

Finally, we define `nm` $e\ D_e := \pi_1$(`nm_rec` $e\ D_e$) and get `nm_spec` $e\ D_e : \mathbb{G}\ e$ (`nm` $e\ D_e$) using the second projection π_2. Extraction of OCaml code from `nm` outputs exactly the algorithm of Fig. 1, illustrating the purely logical (`Prop`) nature of D_e. In order to reason on $\mathbb{D}$/`nm` we show that they satisfy the IR specification given in Fig. 4: the constructors of $\mathbb{D}$ are sufficient to establish the simulated constructors `d_nm_`[012], while `nm_spec` allows us to derive the fixpoint equations of `nm`. Us-

[1] i.e. `g_nm_fun` : $\forall e\ n_1\ n_2, \mathbb{G}\ e\ n_1 \to \mathbb{G}\ e\ n_2 \to n_1 = n_2$.

ing g_nm_fun, we get proof-irrelevance of nm.[2]

```
Inductive Ω : Set := α : Ω | ω : Ω → Ω → Ω → Ω.
Inductive 𝔻 : Ω → Prop :=
  | d_nm_0                    :  𝔻 α
  | d_nm_1 y z                :  𝔻 y → 𝔻 z → 𝔻(ω α y z)
  | d_nm_2 a b c y z D_b D_c   :  𝔻(ω a (nm (ω b y z) D_b)
                                    (nm (ω c y z) D_c))
                              → 𝔻(ω (ω a b c) y z)
with Fixpoint nm e (D_e : 𝔻 e) : Ω := match D_e with
  | d_nm_0                    ↦ α
  | d_nm_1 y z D_y D_z        ↦ ω α (nm y D_y) (nm z D_z)
  | d_nm_2 a b c y z D_b D_c D_a ↦ nm (ω a (nm (ω b y z) D_b)
                                    (nm (ω c y z) D_c)) D_a
end.
```

Figure 4: IR spec. of $\mathbb{D} : \Omega \to \texttt{Prop}$ and $\texttt{nm} : \forall e, \mathbb{D}\ e \to \Omega$.

We show a dependent induction principle for $\mathbb{D}$ (see Fig. 5). The term d_nm_rect states that any dependent property $P : \forall e, \mathbb{D}\ e \to \texttt{Type}$ contains $\mathbb{D}$ as soon as it is closed under the simulated constructors d_nm_[012] of $\mathbb{D}$. The assumption $\forall e\ D_1\ D_2, P\ e\ D_1 \to P\ e\ D_2$ restricts the principle to *proof-irrelevant* properties about the dependent pair (e, D_e). This is exactly what we need to establish properties of nm. Then we can show partial correctness and termination as in [Gie97] – in this example, nm happens to always terminate on a normal form of its input. In a more relational approach, these properties can alternatively be proved using nm_spec and induction on $\mathbb{G}\ x\ n_x$.

```
Theorem d_nm_rect (P : ∀e, 𝔻 e → Type) :
      (∀ e D1 D2, P e D1 → P e D2) → (P _ d_nm_0)
  → (∀ y z D_y D_z, P _ D_1 → P _ D_z → P _ (d_nm_1 y z D_y D_z))
  → (∀ a b c y z D_b D_c D_a, P _ D_b → P _ D_c → P _ D_a …
        … → P _ (d_nm_2 a b c y z D_b D_c D_a))
  → ∀ e D_e, P e D_e.
```

Figure 5: Dependent induction principle for $\mathbb{D} : \Omega \to \texttt{Prop}$.

Though our approach is inspired by IR definitions, in contrast with previous work, e.g. [Bov09], the corresponding principles are established *independently* of any consideration on the semantics or termination of the target function (nm), i.e. without proving any properties of $\mathbb{D}/\texttt{nm}$ a priori. This postpones the study of termination after both $\mathbb{D}$ and nm are defined together with constructors and elimination scheme, fixpoint equations and proof-irrelevance. Moreover, our domain/termination predicate $\mathbb{D}$ is *non-informative*, i.e. it does not carry any computational content. Thus the code obtained by extraction is exactly as intended.

[2] i.e. $\texttt{nm\_pirr} : \forall e\ D_1\ D_2, \texttt{nm}\ e\ D_1 = \texttt{nm}\ e\ D_2$.

Our Coq code is available under a Free Software license [LWM18]. We have successfully implemented other algorithms using the same technique: F91, unification, depth first search as in [Kra10], quicksort, iterations until 0, partial list map as in [BKS16], Huet&Hullot's list reversal [Gie97], etc. The method is not constrained by nested/mutual induction, partiality or dependent types. On the other hand, spotting recursive sub-calls implies the explicit knowledge of all the algorithms that make such calls, a limitation that typically applies to higher order recursive schemes such as e.g. substitutions under binders. Besides growing our bestiary of examples, we aim at formally defining a class of schemes for which our method is applicable, and more practically propose some automation like what is done in Equations [Soz10].

References

[BC05] A. Bove and V. Capretta. Modelling general recursion in type theory. *Math. Struct. Comp. Science*, 15(4):671–708, 2005.

[BKS16] A. Bove, A. Krauss, and M. Sozeau. Partiality and recursion in interactive theorem provers - an overview. *Math. Struct. Comp. Science*, 26(1):38–88, 2016.

[Bov09] A. Bove. Another Look at Function Domains. *Electr. Notes Theor. Comput. Sci.*, 249:61–74, 2009.

[Dyb00] P. Dybjer. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.*, 65(2):525–549, 2000.

[Gie97] J. Giesl. Termination of Nested and Mutually Recursive Algorithms. *J. Autom. Reasoning*, 19(1):1–29, 1997.

[Kra10] A. Krauss. Partial and Nested Recursive Function Definitions in Higher-order Logic. *J. Autom. Reasoning*, 44(4):303–336, 2010.

[LWM18] D. Larchey-Wendling and J.F. Monin. The If-Then-Else normalisation algorithm in Coq. https://github.com/DmxLarchey/ite-normalisation, 2018.

[Set06] A. Setzer. Partial Recursive Functions in Martin-Löf Type Theory. In *CiE 2006*, volume 3988 of *LNCS*, pages 505–515, 2006.

[Soz10] M. Sozeau. Equations: A Dependent Pattern-Matching Compiler. In *ITP 2010*, volume 6172 of *LNCS*, pages 419–434, 2010.

Characterization of eight intersection typed systems *à la* Church*

Luigi Liquori and Claude Stolze

Université Côte d'Azur, Inria, France

This talk is a contribution to the study of the intersection type connective for Church-style λ-calculi. Intersection types are a way to express naturally *ad hoc* polymorphism for pure λ-terms: they also characterize the set of strongly normalizing terms. The difficulty to retain intersection in a Church-style calculus, as type decorations get in the way, is well-known (see [BDS13], page 781, for a quite extensive list of papers on this topic). In our line of work, we encode the type assignment rules of a Curry-style λ-calculus into a Church-style calculus, in a similar way that the Curry-Howard correspondence encode derivation rules of a logic into a Church-style calculus. In particular, we encode intersections as "strong pairs" in the sense of Pottinger and Lopez-Escobar [Pot80, LE85], and also sharing some similarities with virtual tuples of [WDMT02]. The problem is to find an encoding we could consider to be isomorphic such that we get unicity of typing and decidability of type reconstruction.
Starting from the intersection type theories of Coppo-Dezani (CD), Coppo-Dezani-Sallé (CDS), Coppo-Dezani-Venneri (CDV), and Barendregt-Coppo-Dezani (BCD), we define a family of intersection typed systems $\Delta^{\mathcal{T}}_{\mathcal{R}}$ for "Δ-calculi" *à la* Church, parametrized by three things: a binary relation $\mathcal{R}$ on pure λ-terms (which is either $\equiv$ or $=_\beta$), whether the universal type $\mathtt{U}$ is a valid type (if this is the case, we note $\mathtt{U} \in \mathbb{A}$), and an intersection type theory $\mathcal{T}$, whose rules and schemes are the usual ones below:

Minimal rules

(refl) $\sigma \leqslant \sigma$ (incl) $\sigma \cap \tau \leqslant \sigma, \sigma \cap \tau \leqslant \tau$

(glb) $\rho \leqslant \sigma, \rho \leqslant \tau \Rightarrow \rho \leqslant \sigma \cap \tau$ (trans) $\sigma \leqslant \tau, \tau \leqslant \rho \Rightarrow \sigma \leqslant \rho$

Axiom schemes

($\mathtt{U}_{top}$) $\sigma \leqslant \mathtt{U}$ ($\mathtt{U}_{\rightarrow}$) $\mathtt{U} \leqslant \sigma \rightarrow \mathtt{U}$

($\rightarrow\cap$) $(\sigma \rightarrow \tau) \cap (\sigma \rightarrow \rho) \leqslant \sigma \rightarrow (\tau \cap \rho)$

Rule scheme

($\rightarrow$) $\sigma_2 \leqslant \sigma_1, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \rightarrow \tau_1 \leqslant \sigma_2 \rightarrow \tau_2$

Subtyping rules and schemes always verify the minimal rules, and of course the rules ($\mathtt{U}_{top}$) and ($\mathtt{U}_{\rightarrow}$) do not make sense if $\mathtt{U}$ is not a valid type.
Then, we classify the Δ-calculi and we discuss some properties between these Δ-calculi and the original intersection type assignment systems $\lambda^{\mathcal{T}}_{\cap}$, the most important being the property of isomorphism, which states that whenever we assign a type σ to a pure λ-term M, the same type can be assigned to a Δ-term such that the "essence" (defined below) of Δ is M, and conversely, if Δ has type σ, so has the pure λ-term obtained by Δ by applying a suitable essence function. These properties have been thoroughly studied in our technical report [LS18], as well as in our previous works [LR05, LR07, DdLS16].

*Work supported by the COST Action CA15123 EUTYPES "The European research network on types for programming and verification".

The essence function $\wr - \wr$, which relates a Δ-term to a pure λ-term, is defined as follows:

$$\begin{array}{llllll} \wr x \wr & \triangleq & x & \wr \Delta^{\sigma} \wr & \triangleq & \wr \Delta \wr \qquad \wr u_{\Delta} \wr \triangleq \wr \Delta \wr \\ \wr \lambda x{:}\sigma.\Delta \wr & \triangleq & \lambda x.\wr \Delta \wr & \wr \Delta_1\, \Delta_2 \wr & \triangleq & \wr \Delta_1 \wr \wr \Delta_2 \wr \\ \wr \langle \Delta_1\, , \Delta_2 \rangle \wr & \triangleq & \wr \Delta_1 \wr & \wr pr_i\, \Delta \wr & \triangleq & \wr \Delta \wr \qquad i \in \{1,2\} \end{array}$$

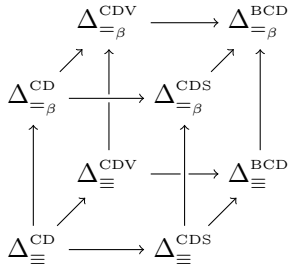
and the main typing rules are:

$$\frac{B \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1 : \sigma \quad B \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \tau \quad \wr \Delta_1 \wr \,\mathcal{R}\, \wr \Delta_2 \wr}{B \vdash^{\mathcal{T}}_{\mathcal{R}} \langle \Delta_1\, , \Delta_2 \rangle : \sigma \cap \tau}\ (\cap I) \qquad \frac{\mathtt{U} \in \mathbb{A}}{B \vdash^{\mathcal{T}}_{\mathcal{R}} u_{\Delta} : \mathtt{U}}\ (top) \qquad \frac{B \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma \quad \sigma \leqslant_{\mathcal{T}} \tau}{B \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta^{\tau} : \tau}\ (\leqslant)$$

where u_{Δ} is a special constant inhabiting $\mathtt{U}$ and Δ^{τ} indicates that Δ is coerced to type τ, and $\langle \Delta_1\, , \Delta_2 \rangle$ indicates an intersection.

Eight type systems for the Δ-calculus can be classified in a Δ-cube where, intuitively, arrows represent system inclusion.

The base system is $\Delta^{\mathrm{CD}}_{\equiv}$, its subtyping relation is the minimal one, $\mathtt{U}$ is not a valid type, and the comparison relation on essences is $\equiv$. We obtain $\Delta^{\mathrm{CDS}}_{\equiv}$ from $\Delta^{\mathrm{CD}}_{\equiv}$ by adding the $\mathtt{U}$ type and the ($\mathtt{U}_{top}$) subtyping rule. $\Delta^{\mathrm{CDV}}_{\equiv}$ does not have the $\mathtt{U}$ type, but its subtyping relation is extended with the rule ($\to\cap$) and the rule scheme ($\to$). $\Delta^{\mathrm{BCD}}_{\equiv}$ has both the $\mathtt{U}$ type and the extended subtyping relation, which satisfy all the previous rules, including ($\mathtt{U}_{\to}$). $\Delta^{\mathrm{CD}}_{=_\beta}$ (respectively $\Delta^{\mathrm{CDS}}_{=_\beta}$,$\Delta^{\mathrm{CDV}}_{=_\beta}$, $\Delta^{\mathrm{BCD}}_{=_\beta}$) is the same system as the one below in the cube where the comparison relation is now $=_\beta$.



All of these intersection typed systems have unicity of type, and all of them except for $\Delta^{\mathrm{CDS}}_{=_\beta}$ and $\Delta^{\mathrm{BCD}}_{=_\beta}$ have decidability of type reconstruction and type checking.

References

[BDS13] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[DdLS16] Daniel J. Dougherty, Ugo de'Liguoro, Luigi Liquori, and Claude Stolze. A realizability interpretation for intersection and union types. In *APLAS*, volume 10017 of *Lecture Notes in Computer Science*, pages 187–205. Springer, 2016.

[LE85] Edgar G. K. Lopez-Escobar. Proof functional connectives. In *Methods in Mathematical Logic*, volume 1130 of *Lecture Notes in Mathematics*, pages 208–221. Springer-Verlag, 1985.

[LR05] Luigi Liquori and Simona Ronchi Della Rocca. Towards an intersection typed system *à la* Church. *Electronic Notes in Theoretical Computer Science*, 136:43–56, 2005.

[LR07] Luigi Liquori and Simona Ronchi Della Rocca. Intersection typed system *à la* Church. *Information and Computation*, 9(205):1371–1386, 2007.

[LS18] Luigi Liquori and Claude Stolze. The Delta-calculus: syntax and types. https://arxiv.org/abs/1803.09660, March 2018.

[Pot80] Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.

[WDMT02] Joe B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Program.*, 12(3):183–227, 2002.

Formal Semantics in Modern Type Theories an Overview

Zhaohui Luo*

Royal Holloway, Univ of London
zhaohui.luo@hotmail.co.uk

I'll give an overview, and report some recent developments, of Formal Semantics in Modern Type Theories (MTT-semantics for short) [25, 14, 4]. MTT-semantics is a semantic framework for natural language, in the tradition of Montague's semantics [21]. However, while Montague's semantics is based on Church's simple type theory [5, 8] (and its models in set theory), MTT-semantics is based on dependent type theories, which we call modern type theories (MTTs),[1] to distinguish them from the simple type theory. Thanks to the recent development, MTT-semantics has become not only a full-blown alternative to Montague's semantics, but also a very attractive framework with a promising future for linguistic semantics.

In this talk, MTT-semantics will be explicated, and its advantages explained, by focussing on the following:

1. The rich structures in MTTs, together with subtyping, make MTTs a nice and powerful framework for formal semantics of natural language.
2. MTT-semantics is both model-theoretic and proof-theoretic and hence very attractive, both theoretically and practically.

By explaining the first point, we'll introduce MTT-semantics and, at the same time, show that the use and development of subtyping [13, 17] play a crucial role in making MTT-semantics viable. The second point, based on [15, 16, 11, 4], shows that MTTs provide a unique and nice semantic framework that was not available before for linguistic semantics. Being model-theoretic, MTT-semantics provides a wide coverage of various linguistic features and, being proof-theoretic, its foundational languages have proof-theoretic meaning theory based on inferential uses[2] (appealing philosophically and theoretically) and it establishes a solid foundation for practical reasoning in natural languages on proof assistants such as Coq [3] (appealing practically). Altogether, this strengthens the argument that MTT-semantics is a promising framework for formal semantics, both theoretically and practically.

References

[1] R. Brandom. *Making It Explicit: Reasoning, Representing, and Discursive Commitment.* Harvard University Press, 1994.

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

[1] By MTTs, we refer to the family of formal systems such as Martin-Löf's intensional type theory (MLTT) [18, 22] in Agda, the type theory CIC_p in Coq [6] and the Unifying Theory of dependent Types (UTT) [12] in Lego/Plastic.

[2] Proof-theoretic semantics, in the sense of [10], has been studied by logicians such as Gentzen [9], Prawitz [24, 23] and Martin-Löf [19, 20] and discussed by philosophers such as Dummett [7] and Brandom [1, 2], among others.

[2] R. Brandom. *Articulating Reasons: an Introduction to Inferentialism*. Harvard University Press, 2000.

[3] S. Chatzikyriakidis and Z. Luo. Natural language reasoning in Coq. *J. of Logic, Language and Information*, 23(4), 2014.

[4] S. Chatzikyriakidis and Z. Luo. *Formal Semantics in Modern Type Theories*. Wiley & ISTE Science Publishing Ltd., 2018. (to appear).

[5] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(1), 1940.

[6] The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.3), INRIA*, 2010.

[7] M. Dummett. *The Logical Basis of Metaphysics*. Duckworth, 1991.

[8] D. Gallin. Intensional and higher-order modal logic: with applications to Montague semantics. 1975.

[9] G. Gentzen. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift*, 39, 1934.

[10] R. Kahle and P. Schroeder-Heister, editors. *Proof-Theoretic Semantics*. Special Issue of Synthese, 148(3), 2006.

[11] G. Lungu. *Subtyping in Signatures*. PhD thesis, Royal Holloway, Univ. of London, 2018.

[12] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[13] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

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

[15] Z. Luo. Formal Semantics in Modern Type Theories: Is It Model-theoretic, Proof-theoretic, or Both? *Invited talk at Logical Aspects of Computational Linguistics 2014 (LACL 2014), Toulouse. LNCS 8535*, pages 177–188, 2014.

[16] Z. Luo. MTT-semantics is model-theoretic as well as proof-theoretic. Manuscript, 2018.

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

[18] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H.Rose and J.C.Shepherdson, editors, *Logic Colloquium'73*, 1975.

[19] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[20] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1), 1996.

[21] R. Montague. *Formal Philosophy*. Yale University Press, 1974. Collected papers edited by R. Thomason.

[22] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[23] D. Prawitz. Towards a foundation of a general proof theory. In P. Suppes *et al.*, editor, *Logic, Methodology, and Phylosophy of Science IV*, 1973.

[24] D. Prawitz. On the idea of a general proof theory. *Synthese*, 27, 1974.

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

The clocks they are adjunctions
Denotational semantics for Clocked Type Theory

Bassel Mannaa and Rasmus Ejlers Møgelberg

IT University of Copenhagen (basm,mogel@itu.dk)

Clocked Type Theory (CloTT)[1] is a new type theory for guarded recursion with multiple clocks. This means in particular that there is a family of modal type constructors $\triangleright^\kappa$ indexed by *clock variables* κ, to be thought of as delays in the sense that $\triangleright^\kappa A$ classifies data of type A delayed one time step on clock κ. The combination of this with a fixed point operator $\mathsf{dfix}^\kappa : (\triangleright^\kappa A \to A) \to \triangleright^\kappa A$, guarded recursive types and universal quantification over clocks allows one to program with coinductive types encoding productivity in types using $\triangleright^\kappa$. Indeed, CloTT has a strongly normalising, confluent reduction semantics satisfying a canonicity result, which proves that for any closed stream definable in CloTT, the nth element can be computed in finite time. Another important application area for CloTT is as a metalanguage for constructing models and operational reasoning principles for advanced programming languages. This talk describes the first denotational model of CloTT [**?**].

Clocked Type theory

The main new contribution of Clocked Type Theory over previous type theories for guarded recursion is the notion of ticks. These are evidence that time has passed, and are used to reason about guarded recursive and coinductive data. In particular, they encode the *delayed substitutions* of [4] and thereby provide computation rules for these. In CloTT the time step modality $\triangleright^\kappa$ is generalised to a form of dependent function type with the introduction and elimination form given by the rules

$$\frac{\Gamma, \alpha : \kappa \vdash_\Delta t : A \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \lambda(\alpha : \kappa).t : \triangleright(\alpha : \kappa).A} \qquad \frac{\Gamma \vdash_\Delta t : \triangleright(\alpha : \kappa).A \qquad \Gamma, \beta : \kappa, \Gamma' \vdash_\Delta}{\Gamma, \beta : \kappa, \Gamma' \vdash_\Delta t\,[\beta] : A\,[\beta/\alpha]}$$

Writing $\triangleright^\kappa A$ for $\triangleright(\alpha : \kappa).A$ when α does not appear free in A, this generalises the modal operator mentioned above. In these rules Δ is a special context for clock variables, and the restriction on the context of t in the elimination rule prevents the same tick from being used to eliminate multiple $\triangleright^\kappa$ in the same term as in $t\,[\beta]\,[\beta]$, which is not well typed, and should not be, because otherwise $\mathsf{dfix}^\kappa(\lambda x.(x\,[\beta]\,[\beta]))$ would inhabit every type of the form $\triangleright^\kappa A$.

Ticks are used in CloTT for reasoning about guarded recursive and coinductive types, but are also used to unfold fixed points. In particular, the term $\mathsf{dfix}^\kappa t\,[\diamond]$ unfolds to $t(\mathsf{dfix}^\kappa t)$. Here $\diamond$ is the *tick constant* with the unusual typing rule

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright(\alpha : \kappa).A \qquad \Gamma \vdash_\Delta \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta (t\,[\kappa'/\kappa])\,[\diamond] : A[\kappa'/\kappa]\,[\diamond/\alpha]}$$

The two substitutions of clock variables in the conclusion prevent typing of terms like $\lambda x.x\,[\diamond]$ which would render the type theory unsound, while maintaining closure of typing under substitutions. The names of ticks control which fixed points are unfolded by the rule mentioned above, and thus play a crucial role in the proof of strong normalisation.

The model

The model is based on a model [5] of the related Guarded Dependent Type Theory [4], but extends it to ticks. The model is done in the category of covariant presheaves on the category whose objects are pairs (Θ, ϑ) of a finite set (of semantic clocks) Θ and a map $\vartheta : \Theta \to \mathbb{N}$ (associating to each clock the time left on the clock), Morphisms $\tau : (\Theta, \vartheta) \to (\Theta', \vartheta')$ in this category are maps $\tau : \Theta \to \Theta'$ such that $\vartheta' \circ \tau \leq \vartheta$ in the pointwise order. Morphisms allow time to tick, but also clocks to be synchronised (if $\tau(\kappa) = \tau(\kappa')$) or introduced (if $\kappa \in \Theta' \setminus \tau[\Theta]$). There is a presheaf of clocks $\mathrm{Clk}(\Theta, \vartheta) = \Theta$, and contexts, types and terms in clock context Δ are modelled in the category $\mathsf{GR}[\Delta]$ of presheaves over the category of elements of Clk^{Δ}. Explicitly, this category of elements has as objects triples (Θ, ϑ, f) where $f : \Delta \to \Theta$.

There appears to be no object of ticks on a clock κ in this model, and thus context extension with ticks $\Gamma, \alpha : \kappa \vdash_\Delta$ cannot be modelled using standard tools. On the other hand, there is a natural model $\blacktriangleright^\kappa$ for the modal type operator $\triangleright^\kappa$ without ticks as an endofunctor on $\mathsf{GR}[\Delta]$. This functor has a left adjoint $\blacktriangleleft^\kappa$. We model context extension as $[\![\Gamma, \alpha : \kappa \vdash_\Delta]\!] = \blacktriangleleft^\kappa [\![\Gamma \vdash_\Delta]\!]$. The functor $\blacktriangleright^\kappa$ extends from contexts to types and terms in the sense that to each semantic context Γ (i.e. object in the presheaf category), and to each semantic type A dependent on Γ there is a semantic type $\blacktriangleright^\kappa A$ dependent on $\blacktriangleright^\kappa \Gamma$. We can use this to interpret the type formation rule

$$\frac{\Gamma, \alpha : \kappa \vdash_\Delta A \text{ type} \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \triangleright(\alpha : \kappa).A \text{ type}}$$

by substituting $\blacktriangleright^\kappa [\![A]\!]$ dependent on $\blacktriangleright^\kappa \blacktriangleleft [\![\Gamma]\!]$ along the unit of the adjunction. This is an example of a dependent right adjoint type [2]. Likewise, modelling $\diamond$ is made complicated by the fact that there is no object of ticks. Nevertheless, it can be modelled using a substitution of appropriate type.

In the singly clocked case (CloTT restricted to the clock context $\Delta = \{\kappa\}$), the left adjoint has a particularly simple description. This fragment can be modelled in the topos of trees [3], modelling a closed type as a family of sets $(X_n)_{n\in\mathbb{N}}$ and maps $(r_n : X_{n+1} \to X_n)_{n\in\mathbb{N}}$. The delay modality is modelled as $(\blacktriangleright\ X)_{n+1} = X_n$, $(\blacktriangleright\ X)_0 = 1$ and the left adjoint is $(\blacktriangleleft\ X)_n = X_{n+1}$. The multiclock setting generalises this idea, but is harder to describe.

References

[1] P. Bahr, H. B. Grathwohl, and R. E. Møgelberg. The clocks are ticking: No more delays! In *LICS*, pages 1–12. IEEE, 2017.

[2] L. Birkedal, R Clouston, B Mannaa, R. E. Møgelberg, A Pitts, and B Spitter. Dependent right adjoint types. Talk submitted to Types, 2018.

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

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

[5] A. Bizjak and R. E. Møgelberg. Denotational semantics for guarded dependent type theory. ArXiv preprint 1802.03744, 2018.

How to define dependently typed CPS using delimited continuations

Étienne Miquey[1]

Équipe Gallinette, INRIA
LS2N, Université de Nantes
etienne.miquey@inria.fr

There are a handful of reasons to pay attention to various flavors of dependently typed translations for dependently typed calculi. For instance, they may allow to ensure the preservation of safeness properties (obtained via dependent types) through a compilation process. Or, as defended by Boulier *et al.* [3], such translations can allow us to define syntactical models for type theories extended with new reasoning principles. In particular, a reasonable plan of attack to allow effectful programs in Coq could be to build successive layers extending CIC with side effects and to justify their soundness by means of syntactic translations. This path has been undertaken recently by Pédrot and Tabareau in [7, 8] to add various classes of effects to CIC. In that perspective, *continuation-passing style* translations give a semantics to control operators, that is to say classical logic, by expliciting the flow of control. Even though it is well-known that classical logic and dependent types can only be mixed if their interactions are restricted, it might still be of great interest to allow more classical reasoning in proof assistants by means of control operators.

In 2002 [2], Barthe and Uustalu first stated the impossibility of CPS translating dependent types. As observed in Bowman *et al.* [4], this result is due to the standard definition of typed CPS translation by double negation. They indeed manage to circumvent this point by using parametric *answer-types* in the translation at the price of an ad-hoc application constructor and of considering an extensional type theory as target language. During this talk, we intend to present a more general method that we introduced to CPS translate a call-by-value sequent calculus with dependent types [5]. Our construction relies on the use of *delimited continuations* in the source language, leading to more parametric answer-types in the translation. The latter turn out to be enough to soundly type the CPS without further addition. We shall now briefly outline the rationale guiding our use of delimited continuations with that respect [5].

It is folklore that sequent calculi are in essence close to the operational semantics of abstract machines, which makes them particularly suitable to define CPS translations. We take advantage of their fine-grained reduction rules to observe the problem already in the source language that we defined as a call-by-value dependently typed calculus. Having a look at the β-reduction rule gives us an insight of what happens. Informally, consider a dependent function $\lambda a.p : \Pi a : A.B$ (*i.e.* p is of type $B[a]$) that is executed in front of a stack $q \cdot e : \Pi a : A.B$ (*i.e.* e is of type $B[q]$). A call-by-value head-reduction rule (in a $\lambda\mu\tilde{\mu}$-like fashion) for this command would then produce a command that we cannot type:

$$\langle \lambda a.p | q \cdot e \rangle \rightsquigarrow \langle q | \tilde{\mu} a.\langle p | e \rangle \rangle \quad \Bigg| \quad \dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \qquad \dfrac{\dfrac{\Gamma, a : A \vdash p : \cancel{B[a]} \mid \Delta \qquad \Gamma, a : A \mid e : \cancel{B[q]} \vdash \Delta}{\langle p | e \rangle : \Gamma, a : A \vdash \Delta}\ \textit{Mismatch}}{\Gamma \mid \tilde{\mu} a.\langle p | e \rangle : A \vdash \Delta}\ (\tilde{\mu})}{\langle q | \tilde{\mu} a.\langle p | e \rangle \rangle : \Gamma \vdash \Delta}\ (\textsc{Cut})$$

The intuition is that in the full command, a has been linked to q at a previous level of the typing judgment. However, the command is still computationally safe, in the sense that the head-reduction imposes that the command $\langle p | e \rangle$ will not be executed before the substitution of a by q is performed. By then, the problem would be solved. This phenomenon can be seen as a desynchronization of the typing process with respect to computation.

Interestingly, the very same happens when trying to define a CPS translation carrying type dependencies. Indeed, a translation of the command above is very likely to look like:

$$[\![q]\!]\,[\![\tilde{\mu}a.\langle p|e\rangle]\!] = [\![q]\!]\,(\lambda a.([\![p]\!]\,[\![e]\!])),$$

where $[\![p]\!]$ is intuitively of type $\neg\neg B[a]$ and $[\![e]\!]$ of type $\neg B[q]$, hence the sub-term $[\![p]\!]\,[\![e]\!]$ is ill-typed.

We follow the idea that the correctness should be guaranteed by a head-reduction strategy, preventing $\langle p|e\rangle$ from reducing before the substitution of a was made. We would like to ensure the same property in the target language, namely that $[\![p]\!]$ cannot be applied to $[\![e]\!]$ before $[\![q]\!]$ has furnished a value to substitute for a. Assuming that q eventually produces a value V, we are informally looking for the following translation and the corresponding reduction sequence:

$$[\![q]\!]\,[\![\tilde{\mu}a.\langle p|e\rangle]\!] \stackrel{?}{=} ([\![q]\!](\lambda a.[\![p]\!]))[\![e]\!] \to ((\lambda a.[\![p]\!])\,[\![V]\!])\,[\![e]\!] \to [\![p]\!][[\![V]\!]/a]\,[\![e]\!]$$

Since $[\![p]\!][[\![V]\!]/a]$ has a type convertible to $\neg\neg B[q]$, the last term is now well-typed.

The first observation is that the term $([\![q]\!](\lambda a.[\![p]\!]))[\![e]\!]$ could be typed by turning the type $A \to \bot$ of the continuation that $[\![q]\!]$ is waiting for into a (dependent) type $\Pi a : A.R[a]$ parameterized by R. This way we could have $[\![q]\!] : \forall R.(\Pi a : A.R[a] \to R[q])$ instead of $[\![q]\!] : ((A \to \bot) \to \bot)$. For $R[a] := (B(a) \to \bot) \to \bot$, the whole term is well-typed. Readers familiar with realizability will also note that such a term is realizable, since it eventually terminates on a correct term $[\![p[q/a]]\!]\,[\![e]\!]$.

The second observation is that such a term suggests the use of delimited continuations [1] to temporarily encapsulate the evaluation of q when reducing such a command. Indeed, the use of delimited continuations allows the source calculus to mimic the aforedescribed reduction:

$$\langle \lambda a.p|q \cdot e\rangle \rightsquigarrow \langle \mu\hat{\mathfrak{p}}.\langle q|\tilde{\mu}a.\langle p|\hat{\mathfrak{p}}\rangle\rangle|e\rangle \rightsquigarrow \langle \mu\hat{\mathfrak{p}}.\langle V|\tilde{\mu}a.\langle p|\hat{\mathfrak{p}}\rangle\rangle|e\rangle \rightsquigarrow \langle \mu\hat{\mathfrak{p}}.\langle p[V/a]|\hat{\mathfrak{p}}\rangle|e\rangle \rightsquigarrow \langle p[V/a]|e\rangle$$

Incidentally, this allows us to introduce a list of dependencies within the typing derivations of judgments involving delimited continuations, and to fully absorb the potential inconsistency in the type of $\hat{\mathfrak{p}}$.

Finally, we shall explain how the translation of dependent sums dually requires co-delimited continuations, how the use of delimited continuations also unveils the need for a restriction to safely use control operators, and how we plan to reuse this method to define a sequent calculus presentation of CIC or even an extension of Munch-Maccagnoni's polarised system L [6] to dependent types.

References

[1] Zena M. Ariola, Hugo Herbelin, and Amr Sabry, *A type-theoretic foundation of delimited continuations*, Higher-Order and Symbolic Computation **22** (2009), no. 3, 233–273.

[2] Gilles Barthe and Tarmo Uustalu, *CPS translating inductive and coinductive types*, Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02), Proceedings, ACM, 2002, pp. 131–142.

[3] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau, *The next 700 syntactical models of type theory*, 6th Conference on Certified Programs and Proofs, CPP 2017, Proceedings, ACM, 2017, pp. 182–194.

[4] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed, *Type-preserving CPS translation of* Σ *and* Π *types is not not possible*, Proc. ACM Program. Lang. **2** (2018), no. POPL, 22:1–22:33.

[5] Étienne Miquey, *A classical sequent calculus with dependent types*, 26th European Symposium on Programming, ESOP 2017, Proceedings (Hongseok Yang, ed.), LNCS, vol. 10201, Springer, 2017, pp. 777–803.

[6] Guillaume Munch-Maccagnoni, *Focalisation and classical realisability*, Computer Science Logic, 23rd international Workshop, CSL 2009, Proceedings, LNCS, vol. 5771, Springer, 2009, pp. 409–423.

[7] Pierre-Marie Pédrot and Nicolas Tabareau, *An effectful way to eliminate addiction to dependence*, 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, IEEE Computer Society, 2017, pp. 1–12.

[8] _____, *Failure is Not an Option - An Exceptional Type Theory*, 27th European Symposium on Programming, ESOP 2018, Proceedings (Amal Ahmed, ed.), LNCS, vol. 10801, Springer, 2018, pp. 245–271.

A type theory for directed homotopy theory

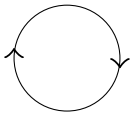
Paige North *

The Ohio State University

Summary. We present rules for a type theory with intended semantics in both higher categories and directed homotopy theory.

Background: directed type theory. Martin-Löf type theory, together with its identity type, is often described as a synthetic theory of ∞-groupoids. Such a description naturally leads one to wonder whether there might be a similar synthetic theory of ∞-categories. Much work has already been done in this direction. In [LH11], the authors construct a type theory that gives a theory of 1-categories. In [RS17], the authors construct a type theory that gives a theory of $(\infty, 1)$-categories. Other works-in-progress ([Nyu15],[War13]) do hope to give a theory of ∞-categories, but are unwieldy for our purpose: namely, to describe a theory not only of higher categories, but also directed homotopy theory.

Background: directed homotopy theory. In brief, directed homotopy theory ([Gra09]) is the study of spaces for which certain paths are singled out and given a direction. For example, one might want to consider the circle S^1 in the category of topological spaces together with the information that the clockwise path shown below is 'allowed', but the analogous counterclockwise path is not 'allowed'.



Such directed spaces appear in many applications, but perhaps most notably in the study of concurrent processes [Faj+16]. Here, a system of concurrent processes is represented by a space whose points represent states of the system and whose directed paths represent possible executions of the system. One can then use a suitable variant of homotopy theory to understand the essential similarities or differences between two such processes.

Directed homotopy theory and higher category theory are very similar. One might describe both of them as the study of objects which have points, undirected paths, and directed paths. The most salient difference between the two, from a categorical perspective, is that while the undirected paths of a higher category are included in its directed paths (since every isomorphism is a morphism), the *directed* paths of a directed space are included in its undirected paths. And thus, the intersection of directed homotopy theory and higher category theory can be understood to be traditional homotopy theory.

A directed type theory. We propose a new type theory with intended semantics in both directed homotopy theory and higher category theory.

We give rules for a *homomorphism* type that are analogous to those given by Martin-Löf for the *identity* type. The notion of direction is thus given as a type, not as a judgment, which distinguishes this type theory from that of [LH11] and [Nyu15].

*Partially supported by AFOSR grant FA9550-16-1-0212.

There are two sorts of morphisms in this type theory: those that preserve identities and those that preserve homomorphisms. Thus there are also two sorts of dependent types: those for which one can perform path induction along elements of the identity type and those for which one can perform path induction along elements of the homomorphism type. This is similar to the type theory of [LH11] and [Nyu15]; but differs from that of [War13] (in particular, it does seem to be possible to perform directed path induction in the system of [War13]).

There is currently an interpretation of this type theory in the category of small categories. In the near future, we hope to complete this research project by providing interpretations in categories of higher categories and categories of directed spaces.

We hope that, in the more distant future, this directed type theory will lay the groundwork for the formalization and automated checking of the mathematics of higher category theory and directed homotopy theory.

References

[Faj+16] L. Fajstrup et al. *Directed Algebraic Topology and Concurrency*. Springer International Publishing, 2016.

[Gra09] M. Grandis. *Directed Algebraic Topology, Models of non-reversible worlds*. Vol. 13. New Mathematical Monographs. Cambridge Univ. Press, 2009.

[LH11] D. R. Licata and R. Harper. "2-Dimensional Directed Type Theory". In: *Electronic Notes in Theoretical Computer Science* 276 (2011), pp. 263 –289.

[Nyu15] A. Nyuts. "Towards a Directed Homotopy Type Theory based on 4 Kinds of Variance". MA thesis. KU Leuven, 2015.

[RS17] E. Riehl and M. Shulman. "A type theory for synthetic ∞-categories". In: *ArXiv e-prints* (2017). arXiv: 1705.07442 [math.CT].

[War13] M. Warren. *Directed Type Theory*. 2013. URL: https://video.ias.edu/univalent/1213/0410-MichaelWarren.

Integers as a Higher Inductive Type

Gun Pinyo and Thorsten Altenkirch*

School of Computer Science, University of Nottingham, UK

We show that the set-truncation condition on the type of integers as a quotient inductive type (QIT)[1], can be reduced to a weaker coherence condition in such a way that it is still a set. This has the advantage that the integers can now be eliminated into not only sets but also more general higher types with this coherence condition.

The integers can easily be implemented as an ordinary inductive type by regarding it as a coproduct of natural numbers. However, in practice, this causes unnecessary complication to the proofs as there are many cases to handle. For example, to define an addition operator, one need to do case analysis on both of left and right integers, each of them has two constructors (as a coproduct), which, in turn as other two constructors (as a natural number), therefore, we will have 8 cases in total. A similar problem applies to a multiplication operator, this results in a complicated proof of something that should be relatively easy such as the distributively law. The problem will be even worse if this version of integers is used to define rational numbers.

Alternatively, the integers are the free group generated over the unit type, so it can be implemented as a QIT as follows[2]:

```
data ℤ^tr : Set where
  zero : ℤ^tr
  succ : ℤ^tr → ℤ^tr
  pred : ℤ^tr → ℤ^tr
  sp : (x : ℤ^tr) → succ (pred x) ≡ x
  ps : (x : ℤ^tr) → pred (succ x) ≡ x
  isSet : (x y : ℤ^tr) → (p q : x ≡ y) → p ≡ q
```

isSet is the set-truncation condition i.e. a condition that forces all paths between two elements of $\mathbb{Z}^{tr}$ to become equal. This condition is necessary. Otherwise, $\mathbb{Z}^{tr}$ would not be a set e.g. both `sp (succ zero)` and[3] `ap succ (ps zero)` have type `succ (pred (succ zero)) ≡ succ zero` but they are not equal to one another. This is a problem (Basold et al. (2017)) because only a set can have a decidable equality due to Hedberg's Theorem (Univalent Foundations Program, 2013, Section 7.2). Therefore, $\mathbb{Z}^{tr}$ without isSet is not the integers we want since it doesn't have the canonicity property, which, in turn, would imply decidable equality.

On the another hand, if a type is set-truncated explicitly then it can only be eliminated into sets. In our case, $\mathbb{Z}^{tr}$ cannot be eliminated into $\mathbb{S}^1$ (Univalent Foundations Program, 2013, Section 6.4) although $\mathbb{Z}^{tr}$ is the fundamental group of it. To fix this, we replace isSet with the following weaker coherence condition:

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

[1] Quotient inductive types are higher inductive type that all of higher path become trivial. See Li (2014), Altenkirch and Kaposi (2016), and Altenkirch et al. (2018)

[2] We use agda-like syntax to present our code.

[3] `ap : ∀{ℓ}{A B : Set ℓ} → (f : A → B) → x ≡ y → f x ≡ f y`

$$\mathsf{coh} \ : \ (\mathsf{x} \ : \ \mathbb{Z}) \ \to \ \mathsf{sp} \ (\mathsf{succ} \ \mathsf{x}) \ \equiv \ \mathsf{ap} \ \mathsf{succ} \ (\mathsf{ps} \ \mathsf{x})$$

and we call these new integers $\mathbb{Z}$. A higher constructor coh looks familiar to the counter-example of $\mathbb{Z}^{tr}$ above (when isSet is removed) nevertheless it is strong enough to imply that $\mathbb{Z}$ is indeed a set. To prove this, let $\mathbb{Z}^{\mathsf{nf}}$ be a type of integers defined as a coproduct of natural numbers, we define functions $\mathsf{nf} \ : \ \mathbb{Z} \ \to \ \mathbb{Z}^{\mathsf{nf}}$ and $\mathsf{emb} \ : \ \mathbb{Z}^{\mathsf{nf}} \ \to \ \mathbb{Z}$ and prove that $\mathsf{emb} \circ \mathsf{nf}$ is equal to the identity function. This proof implies that $\mathbb{Z}$ has a decidable equality, hence, it is a set due to Hedberg's Theorem.

We can understand $\mathbb{Z}$ by observing that all the pred, sp, ps, and coh just express that succ is an equivalence in the sense of *half-adjoint-equivalence* in (Univalent Foundations Program, 2013, Chapter 4). Other alternatives would be to use *bi-invertibility* (having two predecessors) or *contractible-fibres* (stating that each fibre of succ is contractible).

For the formalisation, we use *cubical agda*, an extension of agda inspired by *cubical type theory* (Cohen et al. (2016)). The reason for using this extension is to remove some unnecessary postulates and to avoid irreducible terms.

We hope the idea to replace set truncation by coherence conditions can be applied to many QITs, in particular, the intrinsic syntax of type theory Altenkirch and Kaposi (2016) which also uses set-truncation and hence we are unable to eliminate into a univalent universe. However, this case will be much harder because we would have to represent the coherence conditions for higher type theory explicitly.

Acknowledgements

We would like to thank Paolo Capriotti who suggested this approach.

References

Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices*, 51(1):18–29, 2016.

Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 293–310. Springer International Publishing, 2018.

Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *J. UCS*, 23:63–88, 2017.

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

Nuo Li. *Quotient types in type theory*. PhD thesis, University of Nottingham, July 2014.

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

Embedding Higher-Order Abstract Syntax in Type Theory

Steven Schäfer and Kathrin Stark

Saarland University, Saarbrücken, Germany
{schaefer,kstark}@ps.uni-saarland.de

Higher-order abstract syntax (HOAS) [10] offers a direct representation of higher-order languages, where capture-avoiding substitution is function application, substitution lemmas hold by definition and all derived constructions respect substitution. However, embedding HOAS directly into type theory (and thus profiting from its perks) is difficult, since HOAS relies on an intensional host language function space, while type theoretic function spaces are extensional [6]. We thus propose an indirect approach via embeddings.

An embedding of a HOAS signature into type theory is a term model of the signature. A model gives an interpretation of terms, well-behaved substitutions, and (substitution respecting) recursive definitions on terms. Previous approaches have focused on term representations and renaming - essentially modelling *weak* HOAS [4, 11, 3].

For full models, we extend Hofmann's presheaf semantics for a second-order signature [6], to a construction for arbitrary HOAS signatures (Figure 1). A model consists of a *category of contexts* $\mathbb{D}$ whose morphisms are substitutions and presheaves for every term sort. The action of a substitution on a term gives the notion of instantiation $s[\sigma]$.

The main obstacle to constructing a term model as an inductive type are negative occurrences (tm in lam). Hofmann's insight is that this problem disappears when Tm is representable, i.e., $\mathsf{Tm} = \mathrm{Hom}(\_, T)$ and $\mathbb{D}$ has finite products. In this case, the natural transformation $\mathsf{Lam} : \mathsf{Tm}^{\mathsf{Tm}} \to \mathsf{Tm}$ can equivalently be given by a natural transformation with components $\mathsf{Lam}_X : \mathsf{Tm}(T \times X) \to \mathsf{Tm}(X)$. For every signature, we have to construct a category $\mathbb{D}$.

As a first step, we create the corresponding weak HOAS signature (Figure 1 (b)), in which all non-strictly positive occurrences of a sort (tm) are replaced by a new type v of variables together with a *variable constructor* (var). We build a presheaf model for a weak HOAS signature by constructing a category $\mathbb{C}$ with enough distinct non-terminal objects to represent each sort of variables. In the case of the lambda calculus, any cartesian category with a non-terminal object T will work and we define $V = \mathrm{Hom}(\_, T)$. The canonical choice for $\mathbb{C}$ is the free cartesian category on one object. We model the term sorts by the corresponding inductive types.

We have previously considered presheaf models for weak HOAS signatures [8]. In the present context, the results of [8] show that this construction gives a model for a weak HOAS signature and moreover that such models always extend to models of the corresponding HOAS signature. For terms, this shows that $\mathsf{Tm} : \widehat{\mathbb{C}}$ is a monad relative to V. The construction then extends Tm

$\mathsf{tm} : *$	$\mathsf{Tm} : \widehat{\mathbb{D}}$
$\mathsf{app} : \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm}$	$\mathsf{App} : \mathsf{Tm}^2 \to \mathsf{Tm}$
$\mathsf{lam} : (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}$	$\mathsf{Lam} : \mathsf{Tm}^{\mathsf{Tm}} \to \mathsf{Tm}$

(a) HOAS

$\mathsf{tm}, \mathsf{v} : *$	$\mathsf{Tm}, \mathsf{V} : \widehat{\mathbb{C}}$
$\mathsf{var} : \mathsf{v} \to \mathsf{tm}$	$\mathsf{Var} : \mathsf{V} \to \mathsf{Tm}$
$\mathsf{app} : \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm}$	$\mathsf{App} : \mathsf{Tm}^2 \to \mathsf{Tm}$
$\mathsf{lam} : (\mathsf{v} \to \mathsf{tm}) \to \mathsf{tm}$	$\mathsf{Lam} : \mathsf{Tm}^{\mathsf{V}} \to \mathsf{Tm}$

(b) Weak HOAS

Figure 1: (Weak-) HOAS signatures and presheaf models for the lambda calculus.

to a (representable) presheaf over the Kleisli category of Tm (our chosen $\mathbb{D}$). Additionaly, we obtain a recursion principle with respect to other models of the signature.

The construction of [8] applies to simply typed second-order signatures. We extend this construction to essentially arbitrary HOAS signatures. In the general case, the reduction to a weak HOAS signature can be recursive, the construction has to be stratified in the case of several sorts of terms, and the final relative monad contains more than a single sort of terms. This construction yields several interesting cases, when we consider certain concrete HOAS signatures.

- **Well-typed Terms.** For well-typed terms, the resulting contexts contain a type for each variable. In this case, our construction yields the relative monad of well-typed terms described in [2].

- **Stratified and Mutual Inductive Types.** Applying the construction to signatures with several sorts with binders, such as the types and terms of System F, yields vector parallel substitutions [7, 8]. The category of contexts is the finite product of the respective contexts for all occuring subsorts. In the case of System F we find that the product of the presheaf of types and terms together form a relative monad on the category of contexts.

- **Third-order signatures.** While most practical work has focused on second-order signatures, some applications require higher-order signatures. Consider the $\lambda\mu$-calculus [9, 1], which extends the lambda calculus with an additional constructor.

  $$\mu : ((\mathsf{tm} \to \mathsf{cont}) \to \mathsf{cont}) \to \mathsf{tm}$$

  Our construction yields a weak version with additional variable sort w and variable constructor $\mathsf{var} : \mathsf{w} \to \mathsf{tm} \to \mathsf{cont}$. The final result of our translation corresponds to Parigot's original first-order definition of $\lambda\mu$-terms [9], with a novel notion of instantiation which generalizes the structural substitution of $\lambda\mu$.

- **Type Systems.** Since presheaves can model dependent types we can extend our construction to HOAS predicates, i.e., type systems. The result of the translation is a form of relational typing as in [8], with general "hypothetical" judgments in the context.

  Our translation comes with a suitable notion of substitution on the type system, e.g.:

  $$\frac{\Gamma \vdash s : A \qquad \forall (t : B) \in \Gamma.\ \ \Delta \vdash t[\sigma] : B}{\Delta \vdash s[\sigma] : A}$$

  This corresponds to the notion of *context morphism lemmas* – which in [5] are defined as "parallel substitutions with additonal typing and well-formedness information". Our approach pins down exactly *what* additional information is necessary and identifies context morphism lemmas as the correct notion of substitution on typing judgments.

- **Induction Principles.** Applying the translation to the trivial predicate on terms which states that a term is built from constructors yields a useful induction principle for terms. For the lambda calculus, the induction principle states that given a substitutive predicate $P : \mathsf{tm} \to \mathsf{Prop}$ for which $P\,s \to P\,t \to P(\mathsf{app}\,s\,t)$ and $(\forall t\sigma.\ P\,t \to P\,s[t \cdot \sigma]) \to P(\mathsf{lam}\,s)$ we have $P\,s[\sigma]$ for all terms s and substitutions σ which satisfy $P(\sigma\,i)$ for all i.

Future Work. Ultimately, we are not interested in an arbitrary presheaf model, but in a "free" model. What this should mean is currently an open question. On the practical side we are currently implementing our construction as part of a tool to embed HOAS signatures into Coq.

References

[1] Andreas Abel. A third-order representation of the $\lambda\mu$-calculus. *Electronic Notes in Theoretical Computer Science*, 58(1):97–114, 2001.

[2] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. In *International Conference on Foundations of Software Science and Computational Structures*, pages 297–311. Springer, 2010.

[3] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, volume 43, pages 143–156. ACM, 2008.

[4] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *International Conference on Typed Lambda Calculi and Applications*, pages 124–138. Springer, 1995.

[5] Healfdene Goguen and James McKinna. Candidates for substitution. *LFCS report series-Laboratory for Foundations of Computer Science ECS LFCS*, 1997.

[6] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*, pages 204–213. IEEE, 1999.

[7] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Autosubst 2: Towards reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '17, pages 10–14, New York, NY, USA, 2017. ACM.

[8] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Binder aware recursion over well-scoped de Bruijn syntax. *Certified Programs and Proofs - 7th International Conference, CPP 2018, Los Angeles, USA, January 8-9, 2018*, Jan 2018.

[9] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 190–201. Springer, 1992.

[10] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988.

[11] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(1):87–140, 2008.

Towards Normalization for the Minimal Type Theory

Filippo Sestini

University of Padova, Padova, Italy
sestini.filippo@gmail.com

The Minimalist Foundation is a two-level foundation for constructive mathematics [MS05]. Its intensional level is given by a dependent type theory in the style of Martin-Löf Type Theory, the Minimal Type Theory (mTT) [Mai09]. One peculiarity of its definitional equality is that it rules out the so-called ξ rule, and in general it restricts computation under binders by replacing all compatibility rules with a primitive substitution rule, shown below for the simply-typed case:

$$\frac{\Gamma, x : A \vdash t = s : B}{\Gamma \vdash \lambda x.t = \lambda x.s : A \to B}\ (\xi) \qquad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash a = b : A}{\Gamma \vdash t[a/x] = t[b/x] : B}\ (sub)$$

We observe that the resulting equality corresponds to Çağman and Hindley's notion of *weak reduction* [cH98], which we refer to as *CH-weak reduction*. In this setting, despite the absence of the ξ rule, some form of reduction under binders is still admitted through *sub*. The rule, however, does not suggest in any obvious way an algorithmic method to reduce terms.

Our long-term goal is to find a *constructive* normalization proof for mTT. A *classical* proof is easily found—mTT is normalizing under full β-reduction, that subsumes CH-weak reduction—but it is not sufficient for our purposes, and from it we cannot extract a provably correct normalization algorithm. So far we have shown, and formalized in Agda a proof of normalization for a version of System T with λ-abstractions and CH-weak equality judgments, i.e. with the *sub* rule in place of all compatibility rules, that we call System T^{wk}. We think that the technique used for it can be extended to mTT and dependent types in general.[1] To our knowledge, this is the first analysis of a normalization procedure for a typed λ-calculus under CH-weak reduction. Previous works on similar calculi rejecting the ξ rule focus on weaker notions, like weak *head* reduction [CD97], whose results do not seem to apply to our case.

Recall the definition of *CH-weak reduction* [cH98]: a redex R inside a term P is considered *weak* if and only if no variables that appear free in R are bound in P. A CH-weak contraction is one that affects a weak redex. Under this definition, the term $\lambda x.(\lambda y.x)z$ is, for example, a normal form, whereas $\lambda x.(\lambda y.y)z$ can be reduced to $\lambda x.z$.

The main difficulty with CH-weak evaluation arises from its *relative* nature: whether a redex R is weak depends on what is considered to be its "enclosing" term P. This aspect seems to suggest the necessity to laboriously index every construction involved in the normalization proof with some contextual information regarding P. However, we observe that to recognize a subterm as a weak redex it suffices to be able to distinguish, among its free variables, between those that are free everywhere, and those that are bound somewhere in the enclosing term P, regardless of what P actually is. We call the latter *locally free* variables. A term can thus be CH-weakly reduced by "tagging" its variables accordingly, and contracting only those redexes that are closed w.r.t. locally free variables.

The tagged syntax supports a structurally recursive normalization function on untyped terms, as well as an inductive definition of CH-weak normal forms, with no indexing overhead. The original formal system, however, has no way to express the variable distinction, and the

[1] All results are to be included in the author's forthcoming Master's Thesis, under the supervision of M.E. Maietti.

lack of compatibility rules is at odds with our normalization function, that instead follows the inductive structure of terms. To overcome these issues we employ a type-assignment approach, and extract from the tagged syntax a second calculus (System T^{dwk}) in which the mechanisms of CH-weak reduction are made explicit in the type system. As a result, normalization is easily shown, by instantiating untyped Normalization by Evaluation [Abe13] with the just-mentioned normalization function as interpretation in a syntactic model of CH-weak normal forms.

System T^{dwk} is a calculus with double contexts of the form $\Gamma;\Delta$, keeping track respectively of free variables and *locally free* variables. CH-weak conversion can now be expressed explicitly by reformulating all computation rules so that they only apply to redexes that are provably closed w.r.t. *locally free*, i.e. previously abstracted variables (as in β below.) We can then characterize an equality of the form $\Gamma;\Delta \vdash t = s : A$ intuitively as one where all contracted redexes are *weak*, i.e. do not contain free variables in Δ.

$$\frac{\Gamma; x : A \vdash t : B \qquad \Gamma;\cdot \vdash s : A}{\Gamma;\Delta \vdash (\lambda x.t)s = t[s/x] : B}\ (\beta) \qquad \frac{\Gamma;\Delta, x : A \vdash t = s : B}{\Gamma;\Delta \vdash \lambda x.t = \lambda x.s : A \to B}\ (\xi)$$

The expressive double contexts enable a definition of CH-weak equality judgments in terms of immediate subterms, while avoiding the contraction of non-weak redexes: when proving an equation for a subterm under a binder, it is sufficient to add the bound variable to the second context, hence ruling it out of any redex that will be contracted within the derivation, by construction. We therefore recover all compatibility rules, including a controlled form of ξ rule (shown above), that is of crucial help in establishing soundness of NbE by induction on derivations. In addition, we get admissible substitution rules for both kinds of variables.

We finally show that the "implicit" calculus can be seen as an abstraction over the "explicit" one. In particular, we show that $\Gamma \vdash t = s : A$ is derivable in T^{wk} iff $\Gamma;\cdot \vdash t = s : A$ is derivable in T^{dwk}. Having proved NbE for T^{dwk}, we get normalization for System T^{wk} as a corollary.

Conclusion This work tests a general method to prove normalization with CH-weak equality judgments, and represents a step towards a solution for mTT. The double-context reformulation described above scales to dependent types: we have formalized in Agda a proof of NbE for an "explicit", double-context version of Martin-Löf Type Theory with Π and U. Establishing the correspondence with the "implicit" formulation is left for future work. Besides that, weak forms of reduction have an important role in the study of programming language dynamics, where evaluation is normally limited to programs (i.e., *closed* terms.) We think that also this area could benefit from a better understanding of CH-weak reduction.

References

[Abe13] A. Abel. *Normalization by Evaluation, Dependent Types and Impredicativity.* Habilitation thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2013.

[CD97] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[cH98] Naim Çağman and J.Roger Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1):239 – 247, 1998.

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

[MS05] M.E. Maietti and G. Sambin. Toward a minimalist foundation for constructive mathematics. *From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics*, 48, 2005.

Simulating Codata Types using Coalgebras

Anton Setzer

Dept. of Computer Science, Swansea University, Swansea, UK
a.g.setzer@swan.ac.uk

Abstract

We show how the so called musical notation in Agda for codata types can be considered as syntactic sugar for using codata types in a coalgebraic setting. This allows to simulate codata types using coalgebras while avoiding subject reduction and undecidability problems of codata types. This restricted form of codata types can be added to the coalgebraic setting allowing to shorten proofs and programs involving codata types.

The idea of a codata type is that it is like an algebraic data type (as introduced by the keyword data in Agda/Haskell), but one allows infinite, or more generally non-wellfounded, applications of constructors. An example is the type of colists which in Agda style would be defined as

$$\begin{array}{l}\text{codata coList} : \text{Set where}\\ \quad \begin{array}{lcl}\text{cons} & : & \mathbb{N} \to \text{coList} \to \text{coList}\\ \text{nil} & : & \text{coList}\end{array}\end{array}$$

and which contains finite lists as well as infinite lists such as the list of numbers greater than n, defined as enum $n :=$ cons n (cons $(n+1)$ (cons $(n+2)$ $\cdots$)).
The problems are that when implementing it first in Coq and Agda a subject reduction problem occurred. In our coauthored article [4] we showed that the implicit assumption when pattern matching on codata types, namely that every element of a codata type is introduced by a constructor, results in an undecidable equality.
In order to repair this, in our coauthored article [1] coalgebras and copatterns were proposed for replacing codata types, giving a cleaner theory. They have since been implemented in Agda, In that approach coalgebraic types are defined by their observations. An example is the set of streams which in our desired notation (Agda uses record types instead) would be defined as:

$$\begin{array}{l}\text{coalg Stream} : \text{Set where}\\ \quad \begin{array}{lcl}\text{head} & : & \text{Stream} \to \mathbb{N}\\ \text{tail} & : & \text{Stream} \to \text{Stream}\end{array}\end{array}$$

We can define colists using coalgebras as follows:

$$\begin{array}{l}\text{data coList} : \text{Set where}\\ \quad \begin{array}{lcl}\text{cons} & : & \mathbb{N} \to \infty\text{coList} \to \text{coList}\\ \text{nil} & : & \text{coList}\end{array}\end{array} \qquad \begin{array}{l}\text{coalg } \infty\text{coList} : \text{Set where}\\ \quad \flat \ : \ \infty\text{coList} \to \text{coList}\end{array}$$

The type of colists is called ∞coList which has observation $\flat$, which determines for a colist whether it is of the form nil or (cons n s). coList is the type of colist shapes having these elements. Danielsson [5] pointed out that a key example for codata types is the map function, which can get in variants of codata types quite long definitions. In coalgebras it can be defined as follows:

$$\begin{array}{lcl}\text{map} : (\mathbb{N} \to \mathbb{N}) \to \text{coList} \to \text{coList} & & \\ \text{map } f \ (\text{cons } n \ l) & = & \text{cons } (f \ n) \ (\sharp\text{map } f \ (\flat \ l)))\\ \text{map } f \ \text{nil} & = & \text{nil}\end{array} \qquad \begin{array}{lcl}\sharp\text{map} : (\mathbb{N} \to \mathbb{N}) \to \text{coList} \to \infty\text{coList} & & \\ \flat \ (\sharp\text{map } f \ l) & = & \text{map } f \ l\end{array}$$

In Agda there exists a (currently no longer maintained) variant of codata types, using the so called "Musical Notation" [2], which is a termination checked version of [3]. There one has a

type former $\infty : \mathrm{Set} \to \mathrm{Set}$, which defines a generic coalgebra $\infty\, A$ from A, and an operation $\sharp : A \to \infty\, A$, lifting elements from A to $\infty\, A$. It can be considered as being defined as follows:

$$\begin{array}{ll} \mathrm{coalg}\ (\infty\, A) : \mathrm{Set}\ \mathrm{where} & \sharp : A \to \infty\, A \\ \quad \flat \ : \ \infty\, A \to A & \flat\ (\sharp\ a) = a \end{array}$$

Then colists and the map function are defined as

$$\begin{array}{ll} \mathrm{data}\ \mathrm{coList} : \mathrm{Set}\ \mathrm{where} & \mathrm{map} : (\mathbb{N} \to \mathbb{N}) \to \mathrm{coList} \to \mathrm{coList} \\ \quad \mathrm{cons} \ : \ \mathbb{N} \to \infty\, \mathrm{coList} \to \mathrm{coList} & \mathrm{map}\ f\ (\mathrm{cons}\ n\ l) \ = \ \mathrm{cons}\ (f\ n)\ (\sharp\ (\mathrm{map}\ f\ (\flat\ l))) \\ \quad \mathrm{nil} \ : \ \mathrm{coList} & \mathrm{map}\ f\ \mathrm{nil} \ = \ \mathrm{nil} \end{array}$$

The problem is that the definition of map is, if followed to the letter, non normalising. Furthermore it is not clear, what the right notion of equality is for elements $\sharp\ (\mathrm{map}\ f\ l)$ and $\sharp\ (\mathrm{map}\ f'\ l')$. In addition, $\infty\, A$ cannot defined generically in advance, it needs to be a coalgebra defined simultaneously with A. These problems can be clarified, by understanding them as definitions using coalgebras, and introducing the following notations:

- When introducing a new constant $A : (\vec{x} : \vec{A}) \to \mathrm{Set}$ we define automatically a new constant ∞A of the same type, and when introducing a new function $f : (\vec{y} : \vec{B}) \to A\ \vec{t}$ we define a constant $\sharp f : (\vec{y} : \vec{B}) \to \infty A\ \vec{t}$ with definitions (which are simultaneously defined with the definitions of A and f):

$$\begin{array}{ll} \mathrm{coalg}\ \infty A\ (\vec{x} : \vec{A}) : \mathrm{Set}\ \mathrm{where} & \sharp f : (\vec{y} : \vec{B}) \to \infty A\ \vec{t} \\ \quad \flat \ : \ \infty A\ \vec{x} \to A\ \vec{x} & \flat\ (\sharp f\ \vec{y}) = f\ \vec{y} \end{array}$$

- If A, f are constants, then $\infty\ (A\ \vec{t})$ denotes $\infty A\ \vec{t}$ and $\sharp\ (f\ \vec{t})$ denotes $\sharp f\ \vec{t}$.

With this the above musical definition of coList and map is the same as the previously introduced direct simulation of coList using coalgebras, and the musical notation can be considered as syntactic sugar for simulating codata types in coalgebras. It could live alongside the coalgebraic version, shortening proofs and programs involving codata types.
We will discuss in our talk how to modify this definition to accommodate sized types. One should note that Agda seems to treat mutual coinductive-inductive definitions as $\nu X.\mu Y$ definitions without giving the option of defining them as $\mu Y.\nu X$. Experiments show that this seems to be the case both for the coalgebraic version and the version with musical notation.

References

[1] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In R. Giacobazzi and R. Cousot, editors, *Proceedings of POPL '13*, pages 27–38, New York, NY, USA, 2013. ACM. https://doi.org/10.1145/2429069.2429075.

[2] Agda Wiki. Coinductive data types, 1 January 2011. http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Codatatypes.

[3] T. Altenkirch, N. Danielsson, A. Löh, and N. Oury. ΠΣ: Dependent types without the sugar. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin / Heidelberg, 2010. http://dx.doi.org/10.1007/978-3-642-12251-4_5.

[4] U. Berger and A. Setzer. Undecidability of equality for codata types, February 2018. To appear in proceedings of CMCS'18, available from http://www.cs.swan.ac.uk/~csetzer/articles/CMCS2018/bergerSetzerProceedingsCMCS18.pdf.

[5] N. A. Danielsson. Changes to coinduction, 17 March 2009. Message posted on gmane.comp.lang.agda, available from http://article.gmane.org/gmane.comp.lang.agda/763/.

Automorphisms of Types for Security

Sergei Soloviev[1] and Jan Malakhovski[2]

[1] IRIT, Paul Sabatier University, Toulouse, France,
soloviev@irit.fr
[2] IRIT, Paul Sabatier University, Toulouse, France and ITMO University, Saint-Petersburg, Russia
papers@oxij.org

It is well known that lambda terms may be seen as derived combinators [1]. In this way they may be used to create any program from more elementary building blocks of code. Using them together with automorphisms and isomorphisms (represented by invertible lambda terms) permits to hide (and thus protect) the way how the main program is built from these elementary blocks; if necessary, even the specification may be hidden.

Consider some type S. The closed terms $F : S$ represent combinators that may take other terms as arguments. A possible meaning is that F combines in some way (and we are interested in the case when it is opaque to external users) the operators and data. All the parameters may be fed externally in a controlled way. Let $f_{1 \div n}$ abbreviate $f_1, ..., f_n$.

- If we take $F \equiv \lambda f_{1 \div n} : X \to X \lambda x : X.(f_{\sigma(1)}(...(f_{\sigma(n)}x)...)$ where σ is some permutation of $\{1, ..., n\}$, and apply to some $\phi_{1 \div n}$, F will combine them in any desired order. The $\phi_{1 \div n}$ themselves may be, for example, coding functions, as in [5].
- If we take $F \equiv \lambda f_{1 \div n} : X \to X \lambda x : X.f_i x$ then only one of ϕ will be selected, etc. $\diamond$
- If we take $\Phi \equiv \lambda G : S.G : S \to S$, then we may first apply Φ to some operator, like F considered above, and then "feed" $\phi' s$ (and x in the end).
- The term $\lambda G : S.G$ belongs to $\lambda^1\beta\eta$. In $\lambda^2\beta\eta$ we may add a second-order λ and consider $\Theta \equiv \lambda X.\lambda G : S.G$. In this way the type X also becomes one of controlled parameters, for example it may be Nat, $Bool$ or any other type.
- If dependent types are admitted, the type X itself may depend on terms as parameters.

What may be the role of isomorphisms and automorphisms[1] in a security-oriented picture? We shall describe two possible applications.

Firstly, the type S of the combinator F may have many automorphisms which form a subset of all possible isomorphisms to/from this type. Automorphisms, in difference from isomorphisms, do not change the types of parameters (taken in a fixed order) that F can be applied to. So, if an automorphism θ of S is applied to F, the application $\theta(F)$ to $t_{1 \div n}$ is valid iff the application $Ft_{1 \div n}$ is valid. In difference from automorphisms, an action of an isomorphism θ' (which is not an automorphism) may make invalid the application $\theta'(F)t_{1 \div n}$. This fact may be exploited to detect code transformations performed by third-parties or to execute **zero-knowledge proof** protocols if the distinctions between some type variables remain hidden from external observers.

Secondly, Soloviev [11] had shown that any finite group may be represented as the group of automorphisms of some type in the second order system $\lambda^2\beta\eta$ of [3] and in Z. Luo's typed logical

[1] There exists an extensive literature on isomorphisms of types, see for exemple [3]. Linear isomorphism of types was considered in [9]. Isomorphism in $\lambda^1\beta\eta\pi*$ extended with coproduct (disjunction) was considered in [4]. About isomorphism in dependent type systems, see for example [2, 10]. Automorphisms of types (first studied in [11]) are just isomorphisms of some type to itself, for example the permutations of premises in $A \to A \to B$.

framework LF with dependent products (and in the extensions of these systems). The groups of automorphisms of simple types (the system $\lambda^1\beta\eta$) correspond to the groups of automorphisms of finite trees. This permits to combine the use of λ-terms as combinators and the methods of data protection and cryptography based on group theory.

For illustrative purposes, let us recall the description of the **ElGamal** cryptosystem (cf. [7, 8]). In our case the protocol may use the iterations of a distinguished automorphism $g : A \to A$, where g^m is $g \circ ... \circ g$ (m times).

Private Key: $m, m \in N$.

Public Key: g and g^m.

Encryption. To send a message $a : A$ (in our approach it is not a plain text, but an element of type A, and may have more complex structure) Bob computes g^r and g^{mr} for a random $r \in N$. The ciphertext is $(g^r, g^{mr}a)$.

Decryption. Alice knows m, so if she receives the ciphertext $(g^r, g^{mr}a)$, she computes g^{mr} from g^r, then $(g^{mr})^{-1}$, and then computes a from $g^{mr}a$.

We do not consider here the cryptosystems like **MOR** based on a more sophisticated group theory [8] but we note that they, too, can be represented in type theory using the results of [11].

By encoding a finite cyclic group of prime order as a group of automorphism of some type we can implement ElGamal (or any other cryptographic protocol based on finite groups) since the composition and inverse of type automorphisms (represented by finite hereditary permutations [3]) can be computed in linear time. Clearly, type-based implementations are going to be less efficient than an equivalent long integer-based ones which would make them less desirable for conventional applications. But that alone can make them more desirable for other uses like **proof-of-work** algorithms. Also of note is the fact that the above encryption scheme preserves the structure of $a : A$. Which, for instance, means that Alice needs not typecheck the decrypted a if she trusts Bob to typecheck his.

References

[1] Barendregt, H. (1984) *The Lambda Calculus; Its Syntax and Semantics.* North-Holland Plc.

[2] Delahaye, D. (1999) Information Retrieval in a Coq Proof Library Using Type Isomorphisms. In *TYPES 1999, LNCS*, **1956**, 131-147, Springer-Verlag.

[3] Di Cosmo, R. (1995) *Isomorphisms of types: from lambda-calculus to information retrieval and language design.* Birkhauser.

[4] Fiore, M. Di Cosmo, R., and Balat, V. (2006) Remarks on isomorphisms in typed lambda calculi with empty and sum types. *Annals of Pure and Applied Logic*, **141**(1),35-50.

[5] Gross, T., Modersheim, S. (2011) Vertical protocol composition. CSF Symp., IEEE 24th, 235-250.

[6] Heather, J., Lowe, G., and Schneider, S. (2003) How to prevent type flaw attacks on security protocols. J. of Computer Security, 11(2), 217-244.

[7] Hoffstein, J., Pipher, J. and Silverman, J.H. (2008) An introduction to mathematical cryptography, Undergraduate Texts in Mathematics, Springer, New York, 2008.

[8] Mahalanobis, A. (2015) The MOR cryptosystem and finite p-groups. Contemp. Math., **633**, 81-95.

[9] Soloviev, S.V.(1993) A complete axiom system for isomorphism of types in closed categories. In *A. Voronkov, ed., LPAR'93, LNAI*, **698**, 360-371.

[10] Soloviev, S. (2015) On Isomorphism of Dependent Products in a Typed Logical Framework. *Post-proceedings of TYPES 2014, LIPICS*, **39**, 275-288.

[11] Soloviev, S. (2018) Automorphisms of Types in Certain Type Theories and Representation of Finite Groups. To appear in Math. Structures in Computer Science.

First steps towards proving functional equivalence of embedded SQL

Mirko Spasić and Milena Vujošević Janičić

Faculty of Mathematics, University of Belgrade, Serbia
{mirko | milena} @ matf.bg.ac.rs

SQL is widely adopted and used as a declarative language for accessing databases, e.g. for storing, manipulating and retrieving data. Due to its importance and thanks to development of formal methods, formal analysis of SQL queries attracts more and more attention. For instance, lately, the equivalence of SQL queries, and primarily the query containment problem, have been an area of active research. There are different formally defined semantics of SQL queries: set semantics, bag semantics, bag-set semantics and combined approaches [4], as well as mechanized semantics, called HoTTSQL, based on K-Relations and homotopy type theory [3]. Also, the authors of [2] defined an SQL algebra that provides a semantics for SQL and formalized it within the Coq proof assistant. All of these approaches deal with standalone SQL queries, and the outcome is used by a query optimizer that performs query rewrites. However, SQL queries are most often used within applications, where the database operations are embedded into the programs written in a general purpose imperative programming language. With increasing popularity of embedded devices, with low available memory and critical energy consumption, SQL query performance is important so the embedded SQL approach becomes very popular again [5]. In embedded SQL, statements prefixed with `EXEC SQL` keyword can be intermixed with statements of the used programming language. Automated reasoning about functional equivalence of two versions of such code is very important, for instance in the context of code refactoring, or in the context of minimizing the number of pre-compiled queries [5]. This mix of declarative and imperative semantics makes automated reasoning about such code a challenging problem. To the best of our knowledge, there are no approaches addressing equivalence of two pieces of code in a general purpose programming language with embedded SQL queries.

We propose a system for automated reasoning about SQL embedded into C functions with a goal of checking functional equivalence among two such functions. The overall architecture of the proposed system is given in the Figure 1. Each input function is split into two parts.

SMTc part. This part consists of a C-like function which contains undefined function calls instead of the original SQL queries. This function is then analyzed by our open-source verification tool LAV [6]. LAV uses SMT solving [1] for checking constructed correctness conditions and for the purpose of this work was slightly extended. Undefined function calls introduced by SQL queries are modelled as uninterpreted functions (denoted as *sqlc*) within formulas that LAV generates (named SMTc). Descriptions of these uninterpreted functions are generated by our SQL2SMT tool (formulas SMTsql) and make a link between formulas SMTc and SMTsql.

SMTsql part. This part consists of SQL queries, which are anayzed by the SQL2SMT tool. SQL2SMT generates two kinds of formulas: generic formulas and formulas corresponding to the concrete SQL statemets. The generic formulas introduce types for tables and rows, functions and predicates for describing their connections and Cartesian product axioms for describing table joins. The concrete SQL formulas introduce descriptions of the original SQL queries.

The result of a query may be (i) a single fact, (ii) a tuple or (iii) a sequence of tuples. Let us first assume that the result of a query is a fact, i.e. a number. There are two possible outcomes of execution of any SQL query: there is or there is not relevant data in the database. Execution

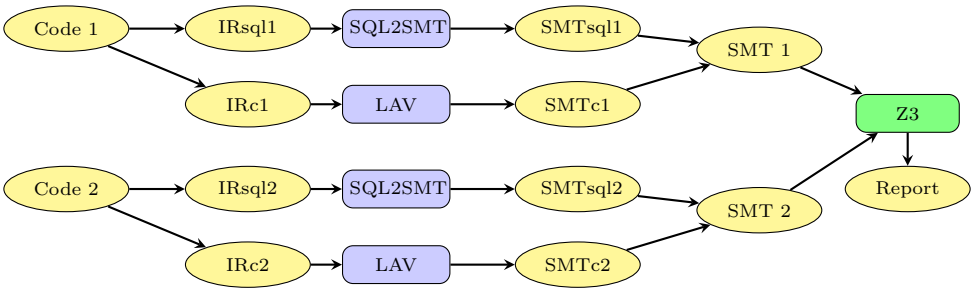


Figure 1: The overall architecture of the system

of the embedded SQL statement returns at the same time whether there are relevant data in the database, and the result itself. We model this as a function *sqlc* with a single return value that depends on the first argument of the function call, being 0 for existence of the relevant data and 1 for the result itself. Formulas describing necessary and sufficient conditions of existence of relevant data, and a value of the result if it exists, are based on the query itself. Transformation of an SQL query to such formulas is based on SQL modified bag-set semantics (bag-set semantics that introduces order over tuples) and the formula is finally rewritten in terms of the theory of bit-vector arithmetic. If result of a query is a tuple, for efficiency reasons, instead of returning it, we introduce a new argument to the function *sqlc*. This argument controls which single value of the tuple is used and it corresponds to the standard SQL projections. Similarly, we introduce one more argument for the case when there are several rows that should be selected as results, corresponding to the cursors in embedded SQL.

In order to prove equivalence of two embedded SQL functions, it is necessary to assume that for specified input conditions there is a unique value for the result in all the selected rows. Otherwise, one cannot be sure which value could be returned by the database management system, and the equivalence cannot and should not be established. Therefore, this assumption is added at the step of constructing a final equivalence formula that is sent to an SMT solver.

The presented approach has been successfully applied on test cases of different sizes and complexities. These examples are publicly available on the LAV page (`http://argo.matf.bg.ac.rs/?content=lav`). The approach is still under development and we work on different kinds of improvements, such as adding support for embedding different SQL data types.

Acknowledgements This work was partially supported by the Serbian Ministry of Science grant 174021 and by COST action CA15123.

References

[1] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. IOS Press, 2009.

[2] V. Benzaken, É. Contejean, C. Keller, and E. Martins. A Coq formalisation of SQL's execution engines. Preprint, 2018.

[3] S. Chu, K. Weitz, A. Cheung, and D. Suciu. Hottsql: Proving query rewrites with univalent sql semantics. *SIGPLAN Not.*, 52(6), 2017.

[4] S. Cohen. Equivalence of queries that are sensitive to multiplicities. *VLDB Journal*, 18(3), 2009.

[5] G. Douglas and R. Lawrence. Improving SQL query performance on embedded devices using pre-compilation. SAC '16, 2016.

[6] M. Vujošević Janičić and V. Kuncak. Development and Evaluation of LAV: An SMT-Based Error Finding Platform. VSTTE '12, Springer, 2012.

HeadREST: A Specification Language for RESTful APIs

Vasco T. Vasconcelos[1], Antónia Lopes[1], and Francisco Martins[2]

LASIGE, University of Lisbon,[1] University of Azores,[2] Portugal

This work is about an industrial application of type theory technology. We have developed HeadREST, a language to model REST services that goes well beyond the descriptive power of current languages. HeadREST is at the basis of an ambitious project addressing fundamental aspects of the API lifecycle; the language aims at supporting testing, validation, runtime monitoring, and code generation.

REST (Representational State Transfer) is an architectural style regarded as an abstract model of the web architecture and based on the concept of resource [3]. According to Fielding, a resource R is a function $M_R(t)$ that associates to each time instant t a set of values, which can be *identifiers* or *representations of resources* [4]. Identifiers are used to distinguish the resource involved in an interaction. Representations capture the current state or the intended state of the resource, and are used to perform actions on the resource.

We focus on REST applications that communicate over HTTP and interact with external systems through web resources identified by Unique Resource Identifiers (URIs). Thus, actions that can be performed on a resource correspond to requests for the execution of the methods offered by HTTP (GET, POST, PUT, and DELETE). The metadata and data to be transferred are sent, respectively, in the header and in the body of the request. As a result to a request a response is produced containing metadata and data to be transferred back to the customer.

Different *interface description languages* (IDLs) have been purposely designed to support the formal description of REST APIs. The most representative are probably Open API Initiative [6] (originally called Swagger), the RESTful API Modeling Language [7] (RAML), and API Blueprint [1]. These IDLs allow a detailed description of the syntactic aspects of the data transferred in REST interactions and are associated to a large number of tools. Being focused on the structure of the data exchanged, they ignore important semantic aspects, including relating different input/output data, the input against the state of the service, and the output against the input.

Our approach is based on two key ideas:

- *Types* to express properties of states and of data exchanged in interactions and
- *Pre- and post-condition* to express the relationship between data sent in requests and that obtained in responses, as well as the resulting state changes.

These ideas are embodied in HeadREST, a language built on the two fundamental concepts of DMinor [2]:

- *Refinement types*, `x:T where e`, consisting of values `x` of type `T` that satisfy property `e`, and
- A *predicate*, `e in T`, which returns true or false depending on whether the value of expression `e` is or is not of type `T`.

HeadREST allows to formally describe properties of data and to observe state changes of REST APIs through a collection of assertions. Assertions take the form of Hoare triples [5]. In $\{\phi\}$ $(a\ t)$ $\{\psi\}$, a is an action (`GET`, `POST`, `PUT`, or `DELETE`), t is an *URI template*, and ϕ and ψ are boolean expressions. Formula ϕ, called the *precondition*, addresses the state in

which the action is performed as well as the data transmitted in the request, whereas ψ, the *postcondition*, addresses the state resulting from the execution of the action together with the values transmitted in the response. The assertion reads

> If a request for the execution of a over an expansion of t carries data satisfying ϕ and the action is performed in a state satisfying ϕ, then the data transmitted in the response satisfies ψ and so does the state resulting from the execution of the action.

A simple contact management system could be based on a *new type*

```
resource Contact
```

There may be many *representations* of such a resource. Here is one:

```
type NameAndEmail = {
  name: (x: string where matches(/^[a-zA-Z]{3,15}$/, x)),
  email: (x: string where contains("@", x))
}
```

An *assertion* describing a successful contact creation could be written as

```
{request in {body: NameAndEmail} &&
 ∀c:Contact. ∀r:NameAndEmail. r repof c ⇒ request.body.name ≠ r.name
}
POST /contacts
{response.code == 200 &&
 response in {body: NameAndEmail, header: {Location: URI}} &&
 ∃c:Contact. response.body repof c && response.header.Location uriof c
}
```

where `request` and `response` are builtin identifiers, and predicates `repof` and `uriof` describe values associated to resources. The precondition asks the new contact name to be unique across all contacts and their representations. In such a case, the postcondition signals success (`code` 200) and states that `response` includes a representation and an URI of the newly created `Contact` resource.

We have used HeadREST to describe different APIs, including a part of GitLab (800 lines of spec code). We have developed an Eclipse plugin to validate HeadREST specifications and a tool to automatically test REST APIs against specifications. These tools rely on an external SMT to solve the semantic subtyping goals required by D-Minor [2]. We are further working on a tool to generate server stubs and client SDKs from HeadREST specifications.

References

[1] API blueprint. https://apiblueprint.org/, 2018. Accessed 25-Jan-2018.

[2] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. *J. Funct. Program.*, 22(1):31–105, 2012.

[3] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[4] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Techn.*, 2(2):115–150, 2002.

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[6] Open API Initiative. https://www.openapis.org, 2018. Accessed 26-Jan-2018.

[7] RESTful API modeling language. https://raml.org, 2018. Accessed 25-Jan-2018.

Experimenting with graded monads: certified grading-based program transformations

Tõnn Talvik[1] and Tarmo Uustalu[2,1]

[1] Dept. of Software Science, Tallinn University of Technology
Akadeemia tee 21B, 12618 Tallinn, Estonia
[2] School of Computer Science, Reykjavik University
Menntavegi 1, 101 Reykjavik, Iceland; tarmo@ru.is

Several authors, above all Katsumata [2], have recently considered type systems that support quantifying the degree of effectfulness of computations via grade-annotated typing judgements and function types. This means revisiting the ideas of type-and-effect systems [3] from static analysis and marrying monads and effects [5] from functional programming, but now employing the new mathematical tool of graded monads or some variation thereof.

A graded monad on a category amounts to a lax monoidal functor from a pomonoid to the category of endofunctors on this base category. Elements of the pomonoid serve as grades of effectfulness (i.e., 'effects' in the terminology of type-and-effect systems). Graded monads are a perfect fit for quantitative analysis of computational effects in a number of theoretical aspects. In practice, however, they require considerable care. It is notoriously easy to slip and come up with candidate examples of pomonoids or graded monads where some required equation fails. How is it then to use them in practice, e.g., in certified programming theory?

In this work, which was mostly accomplished within the master's project of the first author [4], we tried graded monads out with Agda on three notions of effect—exceptions, nondeterministic choice and readable-writable state. Inspired by the work of Benton et al. [1], on program transformations for non-deterministic choice, we implemented the syntax and semantics of a typed call-by-value programming language (with booleans and natural numbers) similar to the computational lambda-calculus and proved correct a number of program equivalences that depend on grading. We also implemented type inference for raw terms, including grade inference.

We considered the following gradings: exceptions—pure, must raise an exception, may raise an exception; nondeterminism—at most n outcomes (also $m..n$ outcomes); state—pure, may read only, may write only, must write, may read and write. We departed from Benton et al. [1] by using logical predicates instead of logical binary relations in the semantics and the correctness proofs of program equivalences. Our type inference (where the emphasis is very much on grade inference) derives the smallest type of a term typable in a given typed context; in our raw terms, lambda-bound variables are type-annotated.

The Agda development consists of the following parts:

- pomonoids (optionally with an added upper semilattice structure),
- pomonoids for grading the effects of exceptions, nondetermistic choice and readable-writable state,
- graded monads,
- graded monads for the effects of exceptions, nondeterministic choice, readable-writable state,
- syntax of the typed language (the generic part and operations specific for each effect considered),
- denotational semantics of the typed language,

- some effect and grading dependent program equivalences, proofs of their correctness,
- syntax of the raw language,
- type inference (including grade inference),
- soundness and completeness of type inference.

A unified approach to different effects should really be based on graded algebraic theories rather than graded monads. With Georgy Lukyanov, the second author has conducted some initial experiments in this direction, which is an unexplored territory also theoretically.

Acknowledgements T. Uustalu was supported by the Estonian Ministry of Education Research institutional research grant no. IUT33-13.

References

[1] N. Benton, A. Kennedy, M. Hofmann, V. Nigam. Counting successes: effects and transformations for non-deterministic programs. In S. Lindley et al., *Wadler Festschrift*, v. 9600 of *Lect. Notes in Comput. Sci.*, pp. 56–72. Springer, 2016.

[2] S.-y. Katsumata. Parametric effect monads and semantics of effect systems. In *Proc. of POPL 2014*, pp. 633–646. ACM, 2014.

[3] J.-P. Talpin, P. Jouvelot. The type and effect discipline. *Inf. and Comput.*, v. 111, n. 2, pp. 245–296, 1994.

[4] T. Talvik. Efektianalüüsidel põhinevate programmiteisenduste sertifitseerimine (Certification of effect analysis based program transformations). MSc thesis. Tallinn Univ. of Technology, 2017.

[5] P. Wadler, P. Thiemann. The marriage of effects and monads. *ACM Trans. on Comput. Log.*, v. 4, n. 1, pp. 1–32, 2003.

Cubical Assemblies and the Independence of the Propositional Resizing Axiom

Taichi Uemura

University of Amsterdam, Amsterdam, the Netherlands
t.uemura@uva.nl

We construct a model of cubical type theory with a univalent and impredicative universe in a category of cubical assemblies. We show that the cubical assembly model does not satisfy the propositional resizing axiom.

We say a universe $\mathcal{U}$ in dependent type theory is *impredicative* if it is closed under arbitrary dependent products: for an arbitrary type A and a function $B : A \to \mathcal{U}$, the dependent product type $\prod_{x:A} B(x)$ belongs to $\mathcal{U}$. An interesting use of such an impredicative universe in homotopy/cubical type theory is the *impredicative encoding* of a higher inductive type [Shu11] as well as ordinary inductive types. For example, in cubical type theory [CCHM18] with an impredicative universe, the unit circle is encoded as

$$S^1 := \prod_{X:\mathcal{U}} \prod_{x:X} \mathsf{Path}(X, x, x) \to X : \mathcal{U}$$

together with a base point $b := \lambda Xxp.x : S^1$, a loop $l := \langle i \rangle.\lambda Xxp.pi : \mathsf{Path}(S^1, b, b)$ and a recursor $r := \lambda Xxps.sXxp : \prod_{X:\mathcal{U}} \prod_{x:X} \mathsf{Path}(X, x, x) \to (S^1 \to X)$. Although the impredicative encoding of a higher inductive type does not satisfy the induction principle in general, some truncated higher inductive types have refinements of the encodings satisfying the induction principle [AFS18].

The first goal of this talk is to present a model of cubical type theory with a univalent and impredicative universe. Since an impredicative universe is modeled in the category of assemblies [LM91] where the impredicative universe classifies *modest families*, our strategy is to construct a model of type theory in the category of cubical objects in assemblies which we will call *cubical assemblies*. There has been a nice set of axioms given by Orton and Pitts [OP16] for modelling cubical type theory without universes of fibrant types in an elementary topos equipped with an interval object $\mathbb{I}$. We will almost entirely follow them, but the category of cubical assemblies is not an elementary topos. So our contribution is to show that the construction given by Orton and Pitts works in a non-topos setting. For constructing the universe of fibrant types, we can use the right adjoint to the exponential functor $(-)^{\mathbb{I}}$ in the same way as Licata, Orton, Pitts and Spitters [LOPS18].

Voevodsky [Voe12] has proposed the *propositional resizing axiom* [Uni13, Section 3.5] which asserts that every homotopy proposition is equivalent to some homotopy proposition in the smallest universe. The propositional resizing axiom can be seen as a form of impredicativity for homotopy propositions. Since the universe in the cubical assembly model is impredicative, one might expect that the cubical assembly model satisfies the propositional resizing axiom. Indeed, for a homotopy proposition A, there is a natural candidate A^* for propositional resizing defined as $A^* := \prod_{X:\mathsf{hProp}}(A \to X) \to X$ together with a function $\eta_A := \lambda aXh.ha : A \to A^*$, where hProp is the universe of homotopy propositions in $\mathcal{U}$. However, the propositional resizing axiom fails in the cubical assembly model. We construct a concrete counterexample to propositional resizing. Note that a homotopy proposition A admits propositional resizing if and only if the function $\eta_A : A \to A^*$ is an equivalence, so it suffices to give a homotopy proposition $\Gamma \vdash A$ such that A does not have an inhabitant but A^* does.

The key fact is that in the category of assemblies, well-supported *uniform* objects are left orthogonal to modest objects [Oos08]. In other words, for any well-supported uniform object A and modest object X, the function $\lambda xa.x : X \to (A \to X)$ is an isomorphism. Uniform objects and modest objects in an internal presheaf category are defined pointwise, and well-supported uniform presheaves are left orthogonal to modest presheaves. Here a presheaf is well-supported if the unique morphism into the terminal presheaf is regular epi, which does not imply the existence of a section. Hence, in order to give a counterexample to propositional resizing, it suffices to find a homotopy proposition $\Gamma \vdash A$ in the cubical assembly model, that is, a presheaf over the category of elements $\int \Gamma$ equipped with the structures of homotopy proposition, that is moreover well-supported and uniform but does not have a section.

For any family A of assemblies over Γ, the codiscrete cubical assembly $\Delta\Gamma \vdash \nabla A$ over the discrete cubical assembly $\Delta\Gamma$ is always a homotopy proposition, and this construction preserves well-supportedness and uniformity. If ∇A has a section, then so does A. Thus, if A is a well-supported and uniform family of assemblies that does not have a section, then the homotopy proposition $\Delta\Gamma \vdash \nabla A$ is a counterexample to propositional resizing. Finally we construct such a family A of assemblies by hand.

References

[AFS18] Steve Awodey, Jonas Frey, and Sam Speight. Impredicative Encodings of (Higher) Inductive Types. In *2018 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2018. to appear. arXiv:1802.02820v1.

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. arXiv:1611.02108v1, doi:10.4230/LIPIcs.TYPES.2015.5.

[LM91] Giuseppe Longo and Eugenio Moggi. Constructive natural deduction and its 'ω-set' interpretation. *Mathematical Structures in Computer Science*, 1(2):215–254, 1991. doi:10.1017/S0960129500001298.

[LOPS18] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal Universes in Models of Homotopy Type Theory. 2018. arXiv:1801.07664v2.

[Oos08] Jaap van Oosten. *Realizability: An Introduction to Its Categorical Side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, San Diego, USA, 2008.

[OP16] Ian Orton and Andrew M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2016.24.

[Shu11] Mike Shulman. Higher inductive types via impredicative polymorphism, 2011. URL: https://homotopytypetheory.org/2011/04/25/higher-inductive-types-via-impredicative-polymorphism/.

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

[Voe12] Vladimir Voevodsky. A universe polymorphic type system. 2012. URL: http://uf-ias-2012.wikispaces.com/file/view/Universe+polymorphic+type+sytem.pdf.

1-Types versus Groupoids

Niels van der Weide, Dan Frumin, and Herman Geuvers

Radboud Univeristy Nijmegen, The Netherlands
nweide@cs.ru.nl, dfrumin@cs.ru.nl, herman@cs.ru.nl

In homotopy type theory, a set is a type A such that for each $x, y : A$ and $p, q : x = y$ we have $p = q$ [9]. In other words, equality of A is proof-irrelevant. A setoid is a type A together with an equivalence relation on A. From a set A we can construct a setoid, namely A with the relation $\lambda xy.\, x = y$. Conversely, from a setoid we can construct a set with quotient types. For a type A and an equivalence relation $R : A \to A \to \textsc{hProp}$, the set quotient A/R can be constructed as a higher inductive type with a point constructor $\mathsf{cl} : A \to A/R$, a path constructor $\mathsf{eqcl} : \prod_{x,y:A} R\ x\ y \to \mathsf{cl}\ x = \mathsf{cl}\ y$, and one saying that A/R is a set [7].

Now let us go to dimension 1. A 1-type is a type A such that for each $x, y : A$ the type $x = y$ is a set. The 1-dimensional version of setoids is groupoids. A groupoid on A consists of a family $G : A \to A \to \textsc{hSet}$, an identity $e : \prod_{x:A} G\ x\ x$, and operations $()^{-1} : \prod_{x,y:A} G\ x\ y \to G\ y\ x$ and $(\cdot) : \prod_{x,y,z:A} G\ x\ y \to G\ y\ z \to G\ x\ z$ satisfying the usual laws for associativity, neutrality, and inverses. We write $\mathsf{grpd}\ A$ for the type of groupoids on A.

Every 1-type A gives rise to a path groupoid $P\ A$ on A defined by $\lambda xy.\, x = y$. However, can we go the other direction? More precisely, given a groupoid G on a type A, our goal is to construct a 1-type $\mathsf{gquot}\ A\ G$ together with a map $\mathsf{gcl} : A \to \mathsf{gquot}\ A\ G$ such that for each $x, y : A$ the types $\mathsf{gcl}\ x = \mathsf{gcl}\ y$ and $G\ x\ y$ are equivalent. To do so, we use the following higher inductive type, which we call the *groupoid quotient*.

Higher Inductive Type $\mathsf{gquot}\ (A : \textsc{Type})\ (G : \mathsf{grpd}\ A) :=$
- $\mid \mathsf{gcl} : A \to \mathsf{gquot}\ A\ G$
- $\mid \mathsf{gcleq} : \prod_{x,y:A} G\ x\ y \to \mathsf{gcl}\ x = \mathsf{gcl}\ y$
- $\mid \mathsf{ge} : \prod_{x:A} \mathsf{gcleq}\ x\ (e\ x) = \mathbf{refl}$
- $\mid \mathsf{ginv} : \prod_{x,y:A} \prod_{g:G\ x\ y} \mathsf{gcleq}\ y\ x\ (g^{-1}) = (\mathsf{gcleq}\ x\ y\ g)^{-1}$
- $\mid \mathsf{gconcat} : \prod_{x,y,z:A} \prod_{g_1:G\ x\ y} \prod_{g_2:G\ y\ z} \mathsf{gcleq}\ x\ z\ (g_1 \cdot g_2) = \mathsf{gcleq}\ x\ y\ g_1 \mathbin{@} \mathsf{gcleq}\ y\ z\ g_2$
- $\mid \mathsf{gtrunc} : \prod_{x,y:\mathsf{gquot}\ A\ G} \prod_{p,q:x=y} \prod_{r,s:p=q} r = s$

To derive the elimination and computation rules of this type, we use the method by Dybjer and Moenclaey [4]. The equations ge, ginv, and gconcat show that gcleq is a homomorphism of groupoids. The constructor gtrunc guarantees that $\mathsf{gquot}\ A\ G$ is a 1-type.

For quotients, the types $\mathsf{cl}\ x = \mathsf{cl}\ y$ and $R\ x\ y$ are equivalent [7]. For $\mathsf{gquot}\ A\ G$, we have a similar statement.

Proposition. *For every $x, y : A$ the types $\mathsf{gcl}\ x = \mathsf{gcl}\ y$ and $G\ x\ y$ are equivalent.*

Each equivalence relation induces a groupoid. The quotient of such an induced groupoid coincides with the set quotient.

Proposition. *If A is a type and R is an equivalence relation on A, then $A/R \simeq \mathsf{gquot}\ A\ \overline{R}$ where $\overline{R}$ is the groupoid induced by R.*

In addition, every 1-type is the groupoid quotient of its path groupoid.

Proposition. *For all 1-types A, we have $A \simeq \mathsf{gquot}\ A\ (P\ A)$ with $P\ A$ the path groupoid on A.*

The category of groupoids has products and coproducts. More precisely, for groupoids $G_1 : \mathsf{grpd}\ A$ and $G_2 : \mathsf{grpd}\ B$, we define $G_1 \times G_2 : \mathsf{grpd}\ (A \times B)$ where the elements are pairs of G_1 and G_2 and the operations are defined pointwise. Similarly, we define $G_1 + G_2 : \mathsf{grpd}\ (A + B)$.

Proposition. *The* gquot *construction commutes with sums and products. More precisely,*

$$\text{gquot } (A \times B)\ (G_1 \times G_2) \simeq \text{gquot } A\ G_1 \times \text{gquot } B\ G_2,$$

$$\text{gquot } (A + B)\ (G_1 + G_2) \simeq \text{gquot } A\ G_1 + \text{gquot } B\ G_2.$$

The proofs of these propositions have been formalized in Coq using the HoTT library [2] and they are available at `https://github.com/nmvdw/groupoids`.

Conclusion and Further Work. Groupoids form a model of type theory [5, 8]. Since 1-types are preserved under dependent products, sums, and identity types [9], they also form a model. This work gives a partial internal comparison between these models.

Dybjer and Moenclaey give an interpretation of higher inductive types in the groupoid model [4]. Internalizing their construction and applying gquot gives an interpretation of 1-HITs as 1-types. The first proposition then characterizes $||x = y||_0$ for this interpretation of higher inductive types.

Another interesting generalization would be using higher groupoids instead of plain groupoids. One can develop a theory of higher groupoids in HoTT similar to the theory of higher groups [3]. This could give a comparison between n-types and n-groupoids. In addition, a full version would be a comparison between ω-groupoids and types [1, 6].

Acknowledgements. We thank Joshua Moerman for giving the inspiration for the last proposition.

References

[1] Thorsten Altenkirch and Ondrej Rypacek. A Syntactical Approach to Weak ω-Groupoids. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012*, pages 16–30, 2012.

[2] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 164–172, New York, NY, USA, 2017. ACM.

[3] Ulrik Buchholtz, Floris van Doorn, and Egbert Rijke. Higher Groups in Homotopy Type Theory. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2018.

[4] Peter Dybjer and Hugo Moeneclaey. Finitary Higher Inductive Types in the Groupoid Model. In *Proceedings of MFPS*, 2017.

[5] Martin Hofmann and Thomas Streicher. The Groupoid Interpretation of Type Theory. *Twenty-five years of constructive type theory (Venice, 1995)*, 36:83–111, 1998.

[6] Peter LeFanu Lumsdaine. Weak ω-Categories from Intensional Type Theory. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, pages 172–187, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] Egbert Rijke and Bas Spitters. Sets in Homotopy Type Theory. *Mathematical Structures in Computer Science*, 25(5):1172–1202, 2015.

[8] Matthieu Sozeau and Nicolas Tabareau. Towards an Internalization of the Groupoid Model of Type Theory. In *Types for Proofs and Programs 20th Meething (TYPES 2014), Book of Abstracts*, 2014.

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

Typing every λ-term with infinitary non-idempotent types

Pierre Vial

Inria (LS2N CNRS)

Infinite types and formulas are known to be unsound, *e.g.*, they allow to type $\Omega = \Delta\,\Delta$ (with $\Delta = \lambda x.x\,x$), the auto-autoapplication and they thus do not ensure any form of normalization/productivity. Moreover, in most infinitary frameworks, it is not difficult to define a type R that can be assigned to *every* λ-term (see below).

A first observation is that one can inhabit every type with $\Omega = \Delta\,\Delta$: let A be *any* formula. We just define the infinite formula F_A by $F_A := (((\dots) \to A) \to A) \to A$ *i.e.* $F_A = F_A \to A$ and we consider:

$$\dfrac{\dfrac{\dfrac{\overline{x : F_A \vdash x : F_A \text{ i.e. } F_A \to A} \quad \overline{x : F_A \vdash x : F_A}}{x : F_A \vdash x\,x : A}}{\vdash \lambda x.x\,x : F_A \to A \text{ i.e. } F_A} \quad \dfrac{\dfrac{\overline{x : F_A \vdash x : F_A} \quad \overline{x : F_A \vdash x : F_A}}{x : F_A \vdash x\,x : A}}{\vdash \lambda x.x\,x : F_A}}{\vdash \Omega : A}$$

Thus, every type A is inhabited by Ω. But given a λ-term t, what types A does t inhabit? A first observation is that every λ-term can easily be typed when infinite types are allowed: let us just define R (standing for "reflexive") by $R = R \to R$. Thus, $R = (R \to R) \to (R \to R) = \dots$ Then, it is very easy to inductively type every term with R. In the inductive steps below, Γ denotes a context that assigns R to each variable:

$$\dfrac{}{\Gamma; x : R \vdash x : R}\,\texttt{ax} \qquad \dfrac{\Gamma; x : R \vdash t : R}{\Gamma \vdash \lambda x.t : R \to R\ (= R)}\,\texttt{abs} \qquad \dfrac{\Gamma \vdash t : R\ (= R \to R) \quad \Gamma \vdash u : R}{\Gamma \vdash t\,u : R}\,\texttt{app}$$

Therefore, every λ-term inhabits the type R. Yet, this does not answer the former question: what types does a term t inhabit? This question has actually no simple answer (we will sketch the reasons why below) and we chiefly focus on one aspect of this problem, namely, the typing constraints caused by the *order* of the λ-terms. Intuitively, the **order of a λ-term** t is its **arity** *i.e.* it is the supremal n such that $t \to_\beta^* \lambda x_1 \dots x_n.u$ (for some term u): the order of t is the number of abstractions that one can output from t. For instance, Ω is of order 0 (it is a **zero term**), the **head normal form (HNF)** $\lambda x_1 x_2.x\,u_1\,u_2\,u_3$ (with u_1, u_2, u_3 terms) is of order 2 and the term $\mathtt{Y}_\lambda := (\lambda x.\lambda y.xx)\lambda x.\lambda y.xx$ (satisfying $\mathtt{Y}_\lambda \to_\beta \lambda y.\mathtt{Y}_\lambda$ and thus, $\mathtt{Y}_\lambda \to_\beta^n \lambda y_1 \dots \lambda y_n.\mathtt{Y}_\lambda$) is of infinite order.

The **order of a type** is the number of its top-level arrows *e.g.*, if o_1, o_2 are *type atoms*, $o_1 \to o_1$, o_1, $o_1 \to o_2 \to o_1$ and $(o_1 \to o_2) \to o_1$ are of respective orders 1, 0, 2, 1. Via the Curry-Howard correspondence, the constructor λx corresponds to the introduction of an implication, and so, in most type systems, a typed term of the form $\lambda x_1 \dots x_n.u$ is typed with an arrow of order $\geqslant n$. For instance, if $\lambda x.\lambda y.u$ is typed, then it is so with a type of the form $A \to B \to C$. Moreover, if a type system satisfies **subject reduction**, the order of B *statically* gives an upper bound to the order of t (static meaning without reduction). A finite *type* has a finite order whereas the finite *term* $\mathtt{Y}_\lambda$ has an infinite order. Yet, unsurprisingly, an infinite type may have an infinite order *e.g.*, R defined by $R = R \to R$ above. This confirms that the typing of any term t with R is trivial and does not bring any information, since the order of a term is of course $\leqslant \infty$.

Intersection type systems (i.t.s.), introduced by Coppo-Dezani [3], generally satisfy subject reduction and **subject expansion**, meaning that typing is stable under anti-reduction. Those systems feature a type constructor $\wedge$ (intersection) and are designed to characterize semantic properties like normalization [6]. From subject expansion and the typing of **normal forms (NF)**, i.t.s. are actually able to *capture* the order of *some* λ-terms. For instance, if an i.t.s. characterizes **head normalization (HN)**, then, every HN term t of order p is typable with a type whose order is also *equal* to p (and not only bounded below by p). Why? Such an i.t.s. usually features arrow types having an empty source (that we generically denote by $\emptyset$), meaning that the underlying functions do not look at their argument. Namely, if $t : \emptyset \to B$, then $t\,u$ is typable with B for *any* term u. This

allows us to easily type any HNF while capturing its order: one just assigns $\emptyset \to \ldots\ldots \emptyset \to o$ (order q) to the head variable x, so that $x\,t_1 \ldots t_q$ is typed with the type atom o and the HNF $\lambda x_1 \ldots x_p.x\,t_1 \ldots t_q$, whose order is p, is typed with an arrow type of order p. Then, by subject expansion, one concludes that every HN term of order p is typable with a type of order p. This method generalizes to other notions of normalization.

Our **main contribution** here is to prove that the order of *every* λ-term can be captured when considering infinite types. We have sketched above the reason why i.t.s. capture the order of the typed terms in the normalizing case: this reduces to typing the "partial" normals forms *i.e.* for a HNF $\lambda x_1 \ldots x_p.x\,t_1 \ldots t_q$, typing just the head variable x, which *cannot be substituted* by a β-reduction step: intuitively, x is **stable**. In contrast, if a variable x is not stable, it may be substituted in a reduction sequence with a term u *i.e.* x is subject to *implicit* and *unforseeable* typing constraints: there is no static way to assign a suitable type to a variable which is not stable in a given term t. Unfortunately, some terms—the so-called **mute terms** [1]—do not ever give rise to stable positions and are thus totally *unproductive*: t is mute iff any reduct of t may be reduced to a redex *e.g.*, Ω is mute. Implicit typing constraints make that there is no canonical technique to type the order of a mute term.

Then, we must actually shift the problem a little: instead of trying to capture the orders of λ-terms, we will actually study typability in a coinductive intersection type system, namely system `S`, that we introduced in [7]. This system takes its name from the **sequences** (families of types indexed by sets of integers) that it uses to represent intersection. System `S` has several features of interest: intersection is not idempotent and it is relevant, meaning that weakening is not allowed. Non-idempotency [5, 4] and relevance make system `S` *resource-aware*. Moreover, system `S` is *rigid*: contrary to most non-idempotent intersection type systems, it features pointers and proof reduction is processed in a deterministic way. Relevance makes the argument at the beginning of this article proving that every term is typable fail *e.g.*, if x does not occur free in t, then $\lambda x.t$ can only have a type of the form $\emptyset \to B$ in system `S`. In particular, in system `S` it is not possible to assign the (non-idempotent counterpart of) the type R satisfying $R = R \to R$ to every abstraction and the induction does not work. On the other hand, the rigidity of system `S` (and the pointers that it gives rise to) enables expressing a notion of **derivation candidate**, which is fundamental to study typability in the unproductive case. The rigidity and the resource-awareness of system `S` makes it the ideal framework to study typability in the infinitary case. We propose a solution to overcome the problem of unproductivity, inspired by first order model theory and resorting to a finite reduction strategy. This constitutes our main technical innovation, which allows us to prove:

Theorem:

- Every term is typable in most infinitary type systems, including the relevant ones.
- Moreover, the order of every term can be captured in coinductive intersection type systems: if t is of order n, then there exists a type B of order n and a context Γ such that $\Gamma \vdash t : B$ is derivable.

As a corollary, our work proves that, in the infinitary relational model [2], every term has a non-empty denotation.

References

[1] A. Berarducci and B. Intrigila. Some new results on easy lambda-terms. *Theor. Comput. Sci.*, 121(1&2), 1993.

[2] A. Bucciarelli, T. Ehrhard, and G. Manzonetto. Not enough points is enough. In *CSL 2007, Lausanne.*

[3] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 4:685–693, 1980.

[4] D. de Carvalho. *Sémantique de la logique linéaire et temps de calcul.* PhD thesis, Université Aix-Marseille, Nov. 2007.

[5] P. Gardner. Discovering needed reductions using type theory. In *TACS 94, Sendai.*

[6] J. Krivine. *Lambda-calculus, types and models.* Masson, 1993.

[7] P. Vial. Infinitary intersection types as sequences: A new answer to Klop's problem. In *LICS, 2017, Reykjavik.*

Using *reflection* to eliminate *reflection*

Théo Winterhalter[1], Matthieu Sozeau[2], and Nicolas Tabareau[1]

[1] Gallinette Project-Team, Inria Nantes France
[2] Pi.R2 Project-Team, Inria and IRIF Paris France

Abstract

Type theories with equality reflection, such as extensional type theory (ETT), are convenient theories in which to formalise mathematics, as they allow to consider provably equal terms as convertible. Although type-checking is undecidable in this context, variants of ETT have been implemented, for example in NuPRL [3] and more recently in Andromeda [2]. The actual objects that can been checked are not proof-terms, but derivations of proof-terms. This suggest that any derivation of ETT can be translated as a typecheckable proof-term of intentional type theory (ITT). However, this result, investigated categorically by Hofmann [5] in 1995, and 10 years later more syntactically by Oury [6], has never given rise to an effective translation. In this paper, we provide the first syntactical translation from ETT to ITT with uniqueness of identity proof and functional extensionality. This translation has been defined and proven correct in Coq [4] and gives rise to an executable plugin that translates a derivation in ETT into an actual Coq typing judgment.

Extensional Type Theory is distinguished from the usual ITT by the reflection rule

$$\frac{\Gamma \vdash_{\mathsf{x}} e : u =_A v}{\Gamma \vdash_{\mathsf{x}} u \equiv v : A}$$

where $u =_A v$ denotes the usual identity type. This setting can be highly practical when considering for instance the type of lists indexed by their length—what we usually call vectors.

```
Inductive vec A : nat -> Type :=
| vnil : vec A 0
| vcons : A -> forall n, vec A n -> vec A (S n).
Arguments vnil {_}. Arguments vcons {_} _ _ _.

Fixpoint rev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m) :=
  match v with vnil => acc
  | vcons a n' v' => rev v' (vcons a m acc)
  end.
```

In this example, Coq will complain that `rev v' (vcons a m acc)` has type `vec A (n' + S m)` whereas `vec A (S n' + m)` was expected. To avoid this, we need to transport along a proof of the equality `n' + S m = S n' + m`. In ETT, thanks to the reflection rule, the terms become convertible and such a definition can be accepted as is.

Eliminating reflection constructively. Hofmann already proved consistency of ETT, but Oury went further in proposing a translation that goes from ETT to ITT (plus Streicher's axiom K [8], functional extensionality and an extra axiom). Our contribution is mainly based on Oury's proof but with small differences that allow us to translate into ITT with axiom K and functional extensionality only. Our main theorem can be summarised as follows, where $\vdash_{\mathsf{i}}$ and $\vdash_{\mathsf{x}}$ denote ITT and ETT judgments respectively, $\sqsubset$ is a syntactical congruence that ignores transports ($t \sqsubset p_*\ \bar{t}$ when $t \sqsubset \bar{t}$), and $\cong$ is heterogenous equality.

Theorem 1 (Translation)**.**

- *If* $\Gamma \vdash_{\mathsf{x}} t : T$ *then for any* $\vdash_{\mathsf{i}} \overline{\Gamma}$ *with* $\Gamma \sqsubset \overline{\Gamma}$ *there exist* $t \sqsubset \bar{t}$ *and* $T \sqsubset \overline{T}$ *such that* $\overline{\Gamma} \vdash_{\mathsf{i}} \bar{t} : \overline{T}$*,*

- *If* $\Gamma \vdash_{\mathsf{x}} u \equiv v : A$ *then for any* $\vdash_{\mathsf{i}} \overline{\Gamma}$ *with* $\Gamma \sqsubset \overline{\Gamma}$ *there exist* $A \sqsubset \overline{A}, A \sqsubset \overline{A}', u \sqsubset \overline{u}, v \sqsubset \overline{v}$ *and* $\overline{e}$ *such that* $\overline{\Gamma} \vdash_{\mathsf{i}} \overline{e} : \overline{u}\ \overline{A} \cong \overline{A}'\ \overline{v}$.

The third axiom is about heterogenous equality and states its congruence with application.

$$\frac{\Gamma \vdash_{\mathsf{i}} p : u_1 \cong u_2 \qquad \Gamma \vdash_{\mathsf{i}} q : v_1 \cong v_2}{\Gamma \vdash_{\mathsf{i}} \mathsf{heqapp}\ p\ q : u_1\ v_1 \cong u_2\ v_2}$$

The way we avoid its use is by having fully annotated terms in both the source and the target: for instance we write $\lambda(x : A).B.t$ for the λ-abstraction (mentioning both the domain and the codomain) and $u\ @_{x:A.B}\ v$ for the application of u to v. This gives us stronger assumptions, in particular we get that the codomains are equal as well. This technicality tends to justify the belief (shared with Andrej Bauer) that the use of reflection asks for a careful handling of terms (while there is no evidence that having unanotated terms can lead to inconsistencies or unwanted results).

One of the other improvements we make is providing evidence that the proof is indeed constructive—which is not clear from the use of existential quantifiers—through a Coq formalisation [7].

TemplateCoq paving the way to a plugin. Our formalisation is done in a setting similar to TemplateCoq [1], a Coq library that allows reflection—this time in the sense of reification—of Coq itself: an inductive represents the internal syntax of Coq and the plugin provides facilities to reify it into actual Coq terms and back. We even provide a (non proven) translation from our internal ITT to TemplateCoq. This means that starting from a derivation in our ETT, we can get a Coq term that corresponds to its translation. Pushing this even further could lead to a plugin in which one *ideally* could write:

```
ETT Fixpoint rev {A n m} (v : vec A n) (acc : vec A m) : vec A (n+m) :=
  match v with vnil => acc
  | vcons a n' v' => rev v' (vcons a m acc)
  end.
Next Obligation. omega. (* Proof that n' + S m = S n' + m *) Qed.
```

The point being, failed conversions would become obligations of equalities, once proven this will produce an ETT derivation that gets translated into a Coq term.

Composing translations. In another direction, this translation also allows for the formalisation of translations that target ETT rather than ITT and still get mechanised proofs of (relative) consistency by composition with this ETT to ITT translation.

References

[1] Abhishek Anand, Simon Boulier, Nicolas Tabareau, and Matthieu Sozeau. Typed Template Coq – Certified Meta-Programming in Coq. In *The Fourth International Workshop on Coq for Programming Languages*, Los Angeles, CA, United States, January 2018.

[2] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Chris Stone. The 'Andromeda' prover. Available at http://www.andromeda-prover.org/.

[3] Robert L. Constable and Joseph L. Bates. The NuPrl system, PRL project. Available at http://www.nuprl.org/.

[4] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2017. Version 8.7, available at http://coq.inria.fr.

[5] Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *International Workshop on Types for Proofs and Programs*, pages 153–164. Springer, 1995.

[6] Nicolas Oury. Extensionality in the calculus of constructions. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2005.

[7] Matthieu Sozeau, Nicolas Tabareau, and Théo Winterhalter. Fork of the 'template-coq' repository. Available at https://github.com/TheoWinterhalter/template-coq.

[8] Thomas Streicher. *Investigations into intensional type theory.* 1993.

Author Index