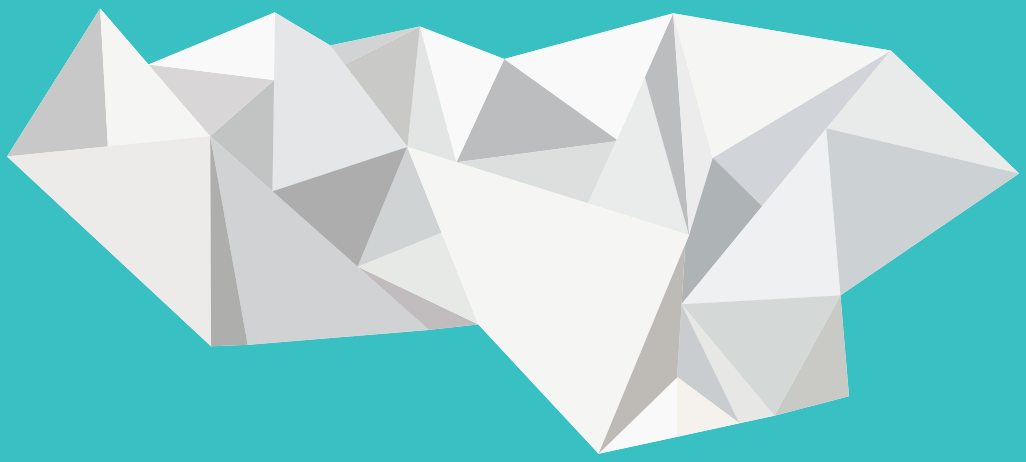
21ST INTERNATIONAL CONFERENCE ON TYPES FOR PROOFS AND PROGRAMS

TYPES 2015

TALLINN | ESTONIA

ABSTRACTS

18-21 MAY



EXCS ESTONIAN COMPUTING

TTÜ KÜBERNEETIKA INSTITUUT
Institute of Cybernetics at TUT

21st International Conference on Types for Proofs and Programs

TYPES 2015

Tallinn, Estonia, 18–21 May 2015

Abstracts

Institute of Cybernetics at Tallinn University of Technology

Tallinn ∘ 2015

21st International Conference on Types for Proofs and Programs
TYPES 2015
Tallinn, Estonia, 18–21 May 2015
Abstracts

Edited by Tarmo Uustalu

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
http://www.ioc.ee/

Cover design by Aive Kalmus

ISBN 978-9949-430-86-4 (print)
ISBN 978-9949-430-87-1 (pdf)

Printed by Multiprint AS

Preface

This volume contains the abstracts of the talks to be given at the *21st International Conference on Types for Proofs and Programs, TYPES 2015*, to take place in Tallinn, Estonia, 18–21 May 2015.

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

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 2015 includes three invited talks by Gilles Barthe (IMDEA Software Institute), Andrej Bauer (University of Ljubljana) and Peter Selinger (Dalhousie University) as well as two tutorials by Joachim Kock (Autonomous University of Barcelona) and Peter LeFanu Lumsdaine (Stockholm University). The contributed part of the programme consists of 35 talks.

Similarly to the 2011, 2013, 2014 editions of the conference, the post-proceedings of TYPES 2015 will appear in Dagstuhl's *Leibniz International Proceedings in Informatics (LIPIcs)* series.

TYPES 2015 is sponsored by the European Regional Development Fund through the CoE project EXCS and ICT R&D project Coinduction.

Tarmo Uustalu

Tallinn, 11 May 2015

Organization

Programme Committee

Andrea Asperti (Università di Bologna)
Robert Atkey (University of Edinburgh / University of Strathclyde)
Ulrich Berger (Swansea University)
Jean-Philippe Bernardy (Chalmers University of Technology)
Edwin Brady (University of St Andrews)
Joëlle Despeyroux (INRIA Sophia Antipolis – Méditerranée)
Herman Geuvers (Radboud Universiteit Nijmegen)
Sam Lindley (University of Edinburgh)
Assia Mahboubi (INRIA Saclay – Île de France)
Ralph Matthes (IRIT, CNRS and Université Paul Sabatier)
Aleksandar Nanevski (IMDEA Software Institute)
Christine Paulin-Mohring (LRI, Université Paris-Sud)
Simona Ronchi Della Rocca (Università di Torino)
Ulrich Schöpp (Ludwig-Maximilians-Universität München)
Bas Spitters (Carnegie Mellon University / Aarhus Universitet)
Pawel Urzyczyn (University of Warsaw)
Tarmo Uustalu (Institute of Cybernetics, Tallinn) (chair)

Organizing Committee

Tiina Laasma, Monika Perkmann, Tarmo Uustalu,
and the Logic and Semantics Group of IoC

Host

Institute of Cybernetics at Tallinn University of Technology

Sponsor

European Regional Development Fund
through the CoE project EXCS and ICT R&D project Coinduction

Table of Contents

Invited talks

Tutorials

Contributed talks

Computer-Aided Cryptography

Gilles Barthe

IMDEA Software Institute, Spain
gilles.barthe@imdea.es

The goal of computer-aided cryptography is to develop tools that help the rigorous design and analysis of cryptographic constructions. Over the years, the focus of computer-aided cryptography has expanded from analysis of cryptographic protocols in the symbolic model to include new goals. Most notably, these goals include analyzing cryptographic primitives and protocols in the computational model, and reasoning about implementations. In this talk, I will illustrate how existing techniques from program verification and program synthesis can be used for achieving a number of goals in computer-aided cryptography.

The Troublesome Reflection Rule

Andrej Bauer

Fakulteta za matematiko in fiziko, Univerza v Ljubljani, Slovenia
andrej.bauer@andrej.com

The equality reflection rule states that propositionally equal terms are judgmentally equal. It is generally frowned upon, as it makes type checking undecidable. Nevertheless, it would be useful to have a workable implementation of a type theory with equality reflection, for instance to tackle Voevodsky's Homotopy Type System. The difficulties created by equality reflection are numerous. It destroys the structural rules of exchange and strengthening, η-reductions cannot be performed without explicit typing annotations in the terms, and injectivity of type constructors becomes an unreasonable expectation. In the talk, I shall discuss the design and implementation of a type theory with reflection that works around these complications while still being a useful tool.

Types for Quantum Computing

Peter Selinger

Department of Mathematics and Statistics, Dalhousie University, Canada
selinger@mathstat.dal.ca

Static type systems are meant to guarantee the absence of certain run-time errors. Quantum computing comes with new kinds of run-time errors, for example attempted violations of the no-cloning property of quantum mechanics. Therefore, quantum computing is potentially interesting from a type-theoretic point of view. In this talk, I will talk about the quantum lambda calculus, a strongly typed functional quantum programming language that has been around for about a decade, and Quipper, a more practical language that is currently implemented as a Haskell EDSL. I will outline some of the challenges involved in creating a type-safe version of Quipper.

Polynomial Functors: A General Framework for Induction and Substitution

Joachim Kock

Departament de Matemàtiques, Universitat Autònoma de Barcelona, Spain
kock@mat.uab.cat

The starting point for this tutorial will be the Seely correspondence between (extensional) dependent type theory and locally cartesian closed categories, such as the category of sets. Under this correspondence, polynomial functors provide semantics for generic data types, and their initial algebras provide semantics for W-types (wellfounded trees). While trees in this approach are defined inductively, it is also possible to define trees combinatorially, directly as certain multivariate polynomial endofunctors. This leads to an alternative characterisation of initial algebras, namely as operations for the free monad on a polynomial endofunctor. The monad itself has important structure not seen at the level of initial algebras, namely the monad multiplication, which always encodes a notion of substitution. I will finish with some outlook regarding the role of polynomial monads (which are essentially operads) to encode and manipulate combinatorial and algebraic structures, and the need for groupoids and higher groupoids to really unleash the power of this formalism—also in intensional type theory.

Higher Inductive Types: What We Understand, What We Don't

Peter LeFanu Lumsdaine

Matematiska institutionen, Stockholms universitet, Sweden
p.l.lumsdaine@gmail.com

HITs were introduced in 2011, following the Oberwolfach mini-workshop on HoTT, as a rather speculative array of new type-constructors, essentially generalising inductive definitions to give (homotopically) higher-dimensional types in the most simple-minded way possible.

Four years later, where do we stand on them? I will survey what we now know about them (or at least, as much of that as I know).

Well-understood: logical consequences. One aspect has worked out as well as anyone could have hoped: their logical consequences within the theory. As mostly developed during the IAS special year, the conjunction of HITs and uxnivalence allows the "synthetic" redevelopment of much of classical abstract homotopy theory within type theory.

Moderately understood: semantics. The semantics has taken longer to get to grips with, but a reasonable understanding is now emerging. The rough ideas are clear, but technicalities related to stability and coherence remain more troublesome than one might expect.

Mike Shulman and I have given models of a restricted range of HITs in a reasonable range of homotopy-theoretic settings (including simplicial sets), using local universes (Lumsdaine, Warren 2015) to obtain stability. On the other hand, Thierry Coquand and collaborators (Bezem, Cohen, Huber, Norsberg) have given models of a wider range of HITs in the cubical set model; and these models are strictly stable by construction, so that the added overhead of local universes is not required.

Little-understood: proof theory. The proof-theoretic aspects of HITs, however, are quite unsatisfactorily understood. Most obviously, we still have no fully general scheme of higher inductive definitions (in the sense of, say, the general inductive definitions of the Calculus of Inductive Constructions). Moreover, even for simple higher inductive types, when taken with judgemental computation rules, we do not understand the effects on proof-theoretic properties such as normalisation, confluence, decidability of typing.

Interesting progress on these issues has appeared recently in the cubical type theories of Altenkirch, Brunerie, Coquand, Licata, and others.

The Next 700 Modal Type Assignment Systems

Andreas Abel

Dept. of Computer Science and Engineering, Gothenburg University, Sweden
abela@chalmers.se

We exhibit a generic modal type system for simply-typed lambda-calculus that subsumes linear and relevance typing, strictness analysis, variance (positivity) checking, and other modal typing disciplines. By identifying a common structure in these seemingly unrelated non-standard type systems, we hope to gain better understanding and a means to combine several analyses into one. This is work in progress.

Our modal type assignment system is parametrized by a (partially) ordered monoid $(P, \_\cdot\_, \mathbb{1}, \leq)$ with a partial, monotone binary operation $\_+\_$ and a default element $p_0 \in P$. Types T, U include at least a greatest type $\top$ and function types $Q \to T$, and form a partial ordering under subtyping $T \leq T'$ with partial meet $T \wedge T'$. Modal types $Q ::= pT$ support composition pQ and partial meet $Q \wedge Q'$ defined by $p(qT) = (pq)T$ and $pT \wedge qT = (p+q)T$. Subtyping $pT \leq p'T'$ holds if $p \leq p'$ and $T \leq T'$.

For typing contexts Γ, Δ, which are total functions from term variables to modal types, modality composition $p\Gamma$, subsumption $\Gamma \leq \Delta$, and meet $\Gamma \wedge \Delta$ are defined pointwise. Finite contexts $x_1 : Q_1, \ldots, x_n : Q_n$ are represented as $\Gamma(x_i) = Q_i$ and $\Gamma(y) = p_0\top$ for $y \neq x_i$.

Judgements $\Gamma \vdash t : T$ and $\Gamma \vdash t : Q$ are given by the following (linear) typing rules:

$$\frac{p \leq \mathbb{1}}{x : pT \vdash x : T}\ \textsc{hyp} \qquad \frac{\Gamma, x : Q \vdash t : T}{\Gamma \vdash \lambda x t : Q \to T}\ \textsc{abs} \qquad \frac{\Gamma \vdash t : Q \to T \qquad \Delta \vdash u : Q}{\Gamma \wedge \Delta \vdash t\,u : T}\ \textsc{app}$$

$$\frac{\Gamma \leq \Delta \qquad \Delta \vdash t : T \qquad T \leq U}{\Gamma \vdash t : U}\ \textsc{sub} \qquad \frac{\Gamma \vdash t : T}{p\Gamma \vdash t : pT}\ \textsc{mod}$$

The default modality p_0 controls weakening: We can use the subsumption rule SUB with $(\Gamma, x : pT) \leq \Gamma$ which holds if $p \leq p_0$ (as then $pT \leq p_0\top = \Gamma(x)$). Meaningful instances of our modal type assignment system abound, here are a few:

1. **Simple typing:** $P = \{1\}$ with $1 + 1 = 1$ and t well-typed if $\Gamma \vdash t : T \neq \top$.

2. **Quantitative typing:** Take some $P \subseteq \mathcal{P}(\mathbb{N})$ closed under $p \cdot q = \{nm \mid n \in p, m \in q\}$ and define $p \leq q$ as $p \supseteq q$ and $p + q$ as $\bigcap\{r \in P \mid r \supseteq \{n + m \mid n \in p, m \in q\}\}$. If $\mathbb{0} := \{0\} \in P$, it is a zero.

   The rule MOD has an intuitive reading in quantitative typing: If t produces a T from resources Γ, we can produce p times T from the p-fold resources $p\Gamma$. Subsumption SUB may allow us to produce less (or the same) from more (or the same) resources. A modal function type $pU \to T$ requires p-fold U to deliver one T.

   Instances of quantitative typing include:

   (a) **Linear typing:** [4] $P = \{\mathbb{0}, \mathbb{1}\}$ with unit $\mathbb{1} = \{1\}$ and default $p_0 = \mathbb{0}$ forbidding weakening with linear variables $x : \mathbb{1}T$ (as $\mathbb{1} \not\leq p_0$). Contraction is also forbidden as $\mathbb{1} + \mathbb{1}$ is undefined.

   (b) **Affine typing:** $P = \{\mathbb{0}, \mathbb{1}\}$ with unit $\mathbb{1} = \{0, 1\}$, allowing weakening as $\mathbb{1} \leq p_0 = \mathbb{0}$.

   (c) **Relevant typing:** $P = \{\mathbb{0}, \mathbb{1}\}$ with unit $\mathbb{1} = \mathbb{N} \setminus \mathbb{0}$, allowing contraction as $\mathbb{1} + \mathbb{1} = \mathbb{1}$.

(d) **Linear and unrestricted hypotheses:** $P = \{!, \mathbb{1}\}$ with $\mathbb{1} = \{1\}$ and $p_0 = ! = \mathbb{N}$. Allows weakening and contraction for $x : !T$.

(e) **Strictness typing:** [2] $P = \{l, s\}$ with *lazy* $p_0 = l = \mathbb{N}$ and unit *strict* $s = \mathbb{N} \setminus \mathbb{0}$. We cannot weaken with strict variables. As $p + q = s$ iff $p = s$ or $q = s$, one strict occurrence of a variable x suffices to classify a function $\lambda xt : sT \to T'$ as strict, whereas a function is lazy only if all occurrences of parameter x are lazy.

3. **Variance (positivity):** $P = \{\emptyset, +, -, \pm\} = \mathcal{P}\{+1, -1\}$ with unit $+ = \{+1\}$ denoting *positive occurrence*, $- = \{-1\}$ *negative occurrence*, $\pm = \{+1, -1\}$ *mixed occurrence*, and $p_0 = \emptyset$ *no occurrence*. With $p \leq q$ iff $p \supseteq q$ and $pq = \{ij \mid i \in p, j \in q\}$ and $p + q = p \cup q$ we obtain *variance typing* aka *positivity checking* for type-level lambda calculi [1].

We can go further and give up the distinction between types and modal types, leading to the types $T, U ::= \top \mid U \to T \mid pT \mid \dots$ quotiented by $p(qT) = (pq)T$. This makes modal types first class, and we can simplify the hypothesis rule to

$$\frac{}{x : T \vdash x : T}\ \text{HYP.}$$

Thus, we subsume further type systems:

1. **Linear typing with exponential:** As 2d, but now $!T$ is a valid type.

2. **Nakano's modality for recursion [3]:** Basic modalities are *later* $\triangleright$ and *always* $\Box$ with $\Box \cdot p = \Box$, generating the modalities $P = \{\triangleright^n, \triangleright^n\Box \mid n \in \mathbb{N}\}$ with unit $\mathbb{1} = \triangleright^0$ and partial order $\triangleright^k\Box \leq \triangleright^l\Box \leq \triangleright^l \leq \triangleright^m$ for $k \leq l \leq m$. Since $x : U \to T$, $y : U \vdash x\,y : T$ entails $x : \triangleright(U \to T), y : \triangleright U \vdash x\,y : \triangleright T$ by MOD, idiomatic application $\lambda x \lambda y.xy : \triangleright(U \to T) \to \triangleright U \to \triangleright T$ is definable.

Acknowledgments. Thanks to the anonymous referees, who helped improving the quality of this abstract through their feedback. This work was supported by Vetenskapsrådet through the project *Termination Certificates for Dependently-Typed Programs and Proofs via Refinement Types*.

References

[1] A. Abel. Polarized subtyping for sized types. *Math. Struct. in Comput. Sci.*, 18(5):797–822, 2008.

[2] S. Holdermans and J. Hage. Making "stricterness" more relevant. *J. of Higher-Order and Symb. Comput.*, 23(3):315–335, 2010.

[3] H. Nakano. A modality for recursion. In *Proc. of 15th Ann. IEEE Symp. on Logic in Computer Science LICS '00*, pp. 255–266. IEEE CS, 2000.

[4] D. Walker. Substructural type systems. In B. C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, ch. 1. MIT Press, 2005.

State and Effect Logics for Deterministic, Non-deterministic, Probabilistic and Quantum Computation

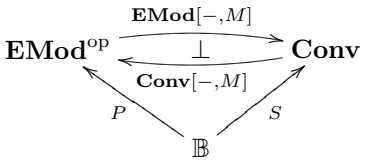
Robin Adams and Bart Jacobs

Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands
robina@cs.ru.nl, bart@cs.ru.nl

We present a type theory that is suitable for reasoning about computations in four very different settings: deterministic, non-deterministic, probabilistic, and quantum computation.

This is possible because, in all cases, a computation can be modelled as a *state transformer* that transforms the state before the computation into the state after. It can also be modelled as a *predicate transformer* that transforms a predicate on the output into the weakest precondition on the input. There is a remarkable correspondence between this picture and the two pictures of quantum theory favored by Schrödinger (who described physical processes in terms of states) and Heisenberg (who used effects) [1].

This correspondence seems to be more than just a coincidence. All four forms of computation can be modelled using three categories in the following arrangement, which we call a *state-and-effect triangle*:



In this structure:

- $\mathbb{B}$ is a category whose objects represent *types* and whose arrows represent *computations*;
- **EMod** is the category of effect modules over an effect monoid M, whose elements represent the *scalars* (probabilities or truth values);
- **Conv** is the category of convex sets over the same effect monoid ([3], see also [2]);
- P and S (for 'predicate' and 'state') are functors which preserve the relevant structure of $\mathbb{B}$ (see below). PA is the effect module of *predicates* over the type A, and SA is the convex set of all *states* of type A.
- There is an adjunction between $\mathbf{EMod}^{\mathrm{op}}$ and **Conv** formed by 'homming' into M ([2]).
- a natural transformation $\models : S \to \mathbf{EMod}[P-, M]$. Given a state $\omega \in SA$ and a predicate $\phi \in PA$, the value $\omega \models \phi$ is an element of M that represents the *probability* or *truth value* that ω satisfies ϕ. In the case of deterministic computation, this will be a Boolean; in our other three examples, it will be a probability in $[0, 1]$.

The four forms of computation are handled by the following four choices of the category $\mathbb{B}$:

The category **Set** for *deterministic* computation, where we model a computation as the function which, given an input

The category $Kl(\mathcal{P}_*)$, the Kleisli category of the non-empty powerset monad $\mathcal{P}_*$, for *non-deterministic* computation, where we model a computation as the function which maps an input to the set of possible outputs.

The category $Kl(\mathcal{D})$, the Kleisli category of the distribution monad $\mathcal{D}$, for *probabilistic* computation, where we model a computation as the function which maps an input to the probability distribution of its outputs.

The category $\mathbf{CStar}^{\mathrm{op}}_{CPU}$, the opposite of the category of C*-algebras and completely positive unital maps, for quantum computation.

Our aim for the future is to give a series of type theories of increasing strength, suitable for the four forms of computation. The final type theory will have these forms of judgement:

- $\Gamma \vdash M : A$, interpreted by the arrows of $\mathbb{B}$;
- $\Gamma \vdash M = N : A$, interpreted by equality of arrows in $\mathbb{B}$;
- $\Gamma \vdash \phi$ eff, which states that ϕ is an effect (predicate) in the context Γ. This will be interpreted by the elements of $P[\![\Gamma]\!]$;
- $\Gamma \vdash \phi \leq \psi$, interpreted by the ordering in $P[\![\Gamma]\!]$.

In this talk, we describe the first two of these theories, which capture the forms of reasoning common to all four models.

Affine Type Theory We first present an *affine type theory*: a type theory in which Contraction does not hold, but Weakening does. Thus, information may be destroyed, but may not be duplicated. (The 'no-cloning theorem' in quantum theory states that a state of a quantum system cannot be duplicated.)

The types of the affine type theory are given by: Type $A ::= X \mid I \mid A \otimes A \mid A + A$.

Affine type theory can be interpreted in all and only the *affine* categories $\mathbb{B}$; i.e. symmetric monoidal categories in which the tensor unit is final, and which have coproducts which the tensor distributes over.

Coproduct Logic We extend the system to *coproduct logic*. Coproduct logic is a type theory that captures structures $(\mathbb{B}, \mathbf{Pred})$ where $\mathbb{B}$ is an affine category, and $\mathbf{Pred} : \mathbb{B} \to \mathbf{Poset}^{\mathrm{op}}$ is a functor which assigns, to each object of $\mathbb{B}$, a poset of *predicates* or *effects*.

In $\mathbf{Set}$ and $Kl(\mathcal{P}_*)$, the predicates on a set A are the subsets of A. In $Kl(\mathcal{D})$, the predicates on A are the 'fuzzy predicates' in $[0,1]^A$. In $\mathbf{CStar}^{\mathrm{op}}_{CPU}$, the predicates on A are the *effects* on A, i.e. the elements $a \in A$ such that $0 \leq a \leq 1$.

Future Work We shall also discuss how comprehension and quotients may be added to these logics, and how this may provide a way of handling measurement with side-effects, as is so important for quantum computation.

References

[1] T. Heinosaari and M. Ziman. *The Mathematical Language of Quantum Theory: From Uncertainty to Entanglement.* Cambridge Univ. Press, 2012.

[2] B. Jacobs. Mesurable spaces and their effect logic. In *Proc. of 28th Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS '13*, pp. 83–92. IEEE CS, 2013.

[3] B. Jacobs. New directions in categorical logic, for classical, probabilistic and quantum logic. arXiv:1205.3940, 2014.

Refinement Types for Algebraic Effects

Danel Ahman and Gordon D. Plotkin

Laboratory for Foundations of Computer Science, University of Edinburgh, United Kingdom

We investigate an algebraic treatment of propositional refinement types for languages with computational effects, and develop a refinement-typed fine-grain call-by-value (FGCBV) [5] language for such refinements. Our work stems from an insight that describing computational effects using algebraic theories $\mathcal{T}_{\mathsf{eff}}$ (following Plotkin and Power [6]) allows us to develop a single framework to account for seemingly different effect specifications found in the literature.

The refinement type system

In our system, *refinement types* τ consist of base types $\mathbf{b}$, product types $\tau_1 \times \tau_2$ and function types $\tau_1 \xrightarrow{\psi} \tau_2$. *Effect refinements* ψ are defined as a fragment of the modal μ-calculus:

$$\psi ::= [\,] \mid \langle \mathsf{op} \rangle(\psi_1, \ldots, \psi_n) \mid \bot \mid \psi_1 \vee \psi_n \mid X \mid \mu X.\psi$$

Effect refinements are intended to represent specifications on computations, in terms of sets of (equivalence classes of) algebraic terms, e.g., the *operation modality* $\langle \mathsf{op} \rangle(\psi_1, \ldots, \psi_n)$ describes a set of algebraic terms $\mathsf{op}(t_1, \ldots, t_n)$ which first perform the computational effect op and then, depending on its outcome, continue as a computation t_i satisfying the corresponding ψ_i.

Effect refinements come with a corresponding subtyping relation $\psi_1 \sqsubseteq \psi_2$, built from rules familiar from modal logic, e.g., $\bot$ is the least element. The relation also includes rules describing the interaction between $\bot$ & $\vee$ and $\langle \mathsf{op} \rangle$, stating that operation modalities are multilinear maps.

We also require the subtyping relation to respect the equations in $\mathcal{T}_{\mathsf{eff}}$. Unfortunately, simply translating the axioms of $\mathcal{T}_{\mathsf{eff}}$ to axioms between corresponding effect refinements is not valid in general: problems arise when the axioms of $\mathcal{T}_{\mathsf{eff}}$ include non-linearity. So instead we limit ourselves to only include derivable *semi-linear equations*, i.e., equations satisfying the conditions in the premise of the rule below. For more discussion about when exactly one can soundly extend algebraic operations on algebras to operations on powersets of algebras, see [1].

$$\frac{\vec{x} \vdash t = u \text{ derivable in } \mathcal{T}_{\mathsf{eff}} \qquad t \text{ linear in } \vec{x} \qquad Vars(u) \subseteq Vars(t)}{t^{\bullet}[\vec{\psi}/\vec{x}] \sqsubseteq u^{\bullet}[\vec{\psi}/\vec{x}]}$$

$(-)^{\bullet}$ is a translation induced by sending operations op to a corresponding modalities $\langle \mathsf{op} \rangle$.

The terms in our system consist of both value terms V and producer terms M, as in FGCBV. Well-typed terms are given by judgments $\Gamma \models_{\mathsf{v}} V : \tau$ and $\Gamma \models_{\mathsf{p}} M : \tau \,!\, \psi$. Here we only present the typing rules that make the interplay between effect refinements and producer terms explicit:

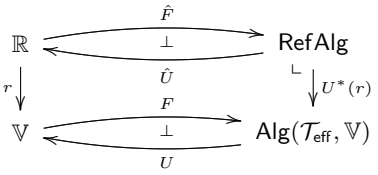
$$\frac{\Gamma \models_{\mathsf{v}} V : \tau}{\Gamma \models_{\mathsf{p}} \mathsf{ret}\, V : \tau \,!\, [\,]} \qquad \frac{\Gamma \models_{\mathsf{p}} M_1 : \tau \,!\, \psi_1 \qquad \ldots \qquad \Gamma \models_{\mathsf{p}} M_n : \tau \,!\, \psi_n}{\Gamma \models_{\mathsf{p}} \mathsf{op}(M_1, \ldots, M_n) : \tau \,!\, \langle \mathsf{op} \rangle(\psi_1, \ldots, \psi_n)} \qquad \frac{\Gamma \models_{\mathsf{p}} M_1 : \tau_1 \,!\, \psi_1 \qquad \Gamma, x : \tau_1 \models_{\mathsf{p}} M_2 : \tau_2 \,!\, \psi_2}{\Gamma \models_{\mathsf{p}} M_1 \,\mathsf{to}\, x : \tau_1 \,\mathsf{in}\, M_2 : \tau_2 \,!\, \psi_1[\psi_2]}$$

Semantics

We give our system a two-level denotational semantics, based on fibred category theory.

We begin by assuming an adjunction model of FGCBV, i.e., a CCC $\mathbb{V}$ (e.g., Set, or ω-Cpo to also accommodate recursion) and a strong adjunction $F \dashv U : \mathsf{Alg}(\mathcal{T}_{\mathsf{eff}}, \mathbb{V}) \to \mathbb{V}$ between $\mathbb{V}$ and the category of $\mathcal{T}_{\mathsf{eff}}$-algebras in $\mathbb{V}$. To model refinement types we assume a CCC $\mathbb{R}$ and a faithful functor $r : \mathbb{R} \to \mathbb{V}$ such that r strictly preserves the CC structure on $\mathbb{R}$, r is a partially-ordered bifibration, and r has fibre-wise small coproducts that are preserved by reindexing functors.

To model effect refinements, we construct a separate bifibration $U^*(r) : \mathsf{RefAlg} \to \mathsf{Alg}(\mathcal{T}_{\mathsf{eff}}, \mathbb{V})$, by applying change of base to r along U. We interpret effect refinements $\Delta \vdash \psi$ as functors $[\![\psi]\!]_{\mathcal{A}} : \mathsf{RefAlg}_{\mathcal{A}} \times \overrightarrow{\mathsf{RefAlg}_{\mathcal{A}}} \to \mathsf{RefAlg}_{\mathcal{A}}$, separately for each algebra $\mathcal{A}$ from $\mathsf{Alg}(\mathcal{T}_{\mathsf{eff}}, \mathbb{V})$. For closed effect refinements $\vdash \psi$, the interpretations for different algebras extend to a single endofunctor $[\![\psi]\!] : \mathsf{RefAlg} \to \mathsf{RefAlg}$ that preserves the underlying algebras, i.e., $U^*(r) \circ [\![\psi]\!] = U^*(r)$.



$[\![\vdash \tau : \mathsf{Ref}(A)]\!] \in ob(\mathbb{R})$ such that $r([\![\tau]\!]) = [\![A]\!]$

$[\![\Gamma \models_{\mathbb{V}} V : \tau]\!] : [\![\Gamma]\!] \longrightarrow [\![\tau]\!]$

$[\![\Gamma \models_{\mathbb{P}} M : \tau \,!\, \psi]\!] : [\![\Gamma]\!] \longrightarrow (\hat{U} \circ [\![\psi]\!] \circ \hat{F})([\![\tau]\!])$

Notice that the interpretation makes use of the adjunction $\hat{F} \dashv \hat{U}$, induced by the change of base along U [2, Cor. 3.2.5]. Importantly for us, $\hat{F} \dashv \hat{U}$ sits over $F \dashv U$ in an appropriate sense.

Applications

For example, we can represent *effect annotations* ε from type-and-effect systems as suitable fixed point refinements. Following Kammar and Plotkin [3], we can take ε to consist of a set of algebraic operations (the effects permitted to occur in the program) and define the corresponding effect refinement as $\psi_\varepsilon \stackrel{\text{def}}{=} \mu X.\, [\,] \vee \bigvee_{\mathsf{op} \in \varepsilon} \langle \mathsf{op} \rangle (X, \ldots, X)$. In addition, we can equip our system with a relational semantics and validate effect-dependent optimizations, again following [3], based on the algebraic properties determined by effect refinements ψ. However, as our refinements also take the temporal structure of computation into account, we can further validate more involved optimizations, such as dead-code elimination in stateful computation.

Our other examples include *Hoare refinements* corresponding to Hoare types from Hoare Type Theory, and *protocol refinements* for I/O, similar to session types and trace effects.

Related work

The literature contains a range of work on modeling type-and-effect systems by various forms of indexed monad-like structures. Our system is closest to that of Katsumata [4] whose parametric effect monads are indexed by ordered monoids. A fundamental difference with our system is that, rather than taking an abstract indexed-monad view of effect annotations, we obtain both as derived constructs within a wider algebraic theory of effects. In particular, the corresponding parametric effect monad is obtained with the monoid given by closed effect refinements ψ.

Acknowledgments We thank O. Kammar, S. Katsumata, B. Kavanagh, S. Lindley, J. McKinna, A. Simpson, T. Uustalu, and P. Wadler for many useful discussions.

References

[1] N. D. Gautam. The validity of equations of complex algebras. *Arch. für Math. Logik und Grundl. der Math.*, 3(3–4):117–124, 1957.

[2] C. Hermida. *Fibrations, Logical Predicates and Indeterminates.* PhD thesis, U. of Edinburgh, 1993.

[3] O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *Proc. of POPL '12*, pp. 349–360. ACM, 2012.

[4] S. Katsumata. Parametric effect monads and semantics of effect systems. In *Proc. of POPL '14*, pp. 633–646. ACM, 2014.

[5] P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Inform. and Comput.*, 185(2):182–210, 2003.

[6] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proc. of FoSSaCS '02*, v. 2303 of *Lect. Notes in Comput. Sci.*, pp. 342–356. Springer, 2002.

Non-wellfounded Trees in Homotopy Type Theory*

Benedikt Ahrens[1], Paolo Capriotti[2] and Régis Spadotti[1]

[1] Institut de Recherche en Informatique de Toulouse, Université Paul Sabatier, France
[2] School of Computer Science, University of Nottingham, United Kingdom
{ahrens,spadotti}@irit.fr, pvc@cs.nott.ac.uk

Coinductive data types are used in functional programming to represent infinite data structures. Examples include the ubiquitous data type of streams over a given base type, but also more sophisticated types.

From a categorical perspective, coinductive types are characterized by a *universal property*, which specifies the object with that property *uniquely* in a suitable sense. More precisely, a coinductive type is specified as the *terminal coalgebra* of a suitable endofunctor. In this category-theoretic viewpoint, coinductive types are dual to *inductive* types, which are defined as initial algebras.

Inductive, resp. coinductive, types are usually considered in the principled form of the family of W-types, resp. M-types, parametrized by a type A and a dependent type family B over A, that is, a family of types $(B(a))_{a:A}$. Intuitively, the elements of the coinductive type $\mathsf{M}(A, B)$ are trees with nodes labeled by elements of A such that a node labeled by $a : A$ has $B(a)$-many subtrees, given by a map $B(a) \to \mathsf{M}(A, B)$; see Figure 1 for an example. The *inductive* type $\mathsf{W}(A, B)$ contains only trees where any path within that tree eventually leads to a *leaf*, that is, to a node $a : A$ such that $B(a)$ is empty.

$a, b, c : A$
$B(a) = 0$
$B(b) = 2$
$B(c) = 3$

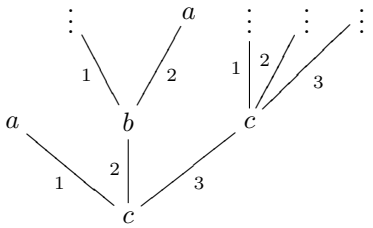


Figure 1: Example of a tree (adapted from [7])

The present work takes place in intensional Martin-Löf type theory extended by Voevodsky's *Univalence Axiom*. We show that, in this type theory, *coinductive* types in the form of M-types can be *derived* from *inductive* types. (More precisely, only one specific W-type is needed: the type of natural numbers, which is readily specified as a W-type [4].) Indeed, given a signature (A, B) specifying a shape of trees as described above, we construct the M-type associated to that signature and prove its universal property. The construction can be seen as a higher-categorical analogue of the classical construction of the terminal coalgebra of some endofunctor as the limit of a chain.

The result presented in this work is not surprising: indeed, the constructibility of coinductive types from inductive types has been shown in *extensional* type theory (that is, type theory with

*The work of Benedikt Ahrens was partially supported by the CIMI (Centre International de Mathmatiques et d'Informatique) Excellence program ANR-11-LABX-0040-CIMI within the program ANR-11-IDEX-0002-02 during a postdoctoral fellowship.

identity reflection) [7, 1], as well as in type theory satisfying Axiom K [3]. It was conjectured to work in homotopy type theory, that is, the type theory described in [6], during a discussion on the HoTT mailing list [5].

We have formalized our results in the proof assistant Agda.

The theorem we prove here is actually more general than described above: instead of plain M-types as described above, we construct *indexed* M-types, which can be considered as a form of "simply-typed" trees, typed over a type of indices I. Plain M-types then correspond to the mono-typed indexed M-types, that is, to those for which $I = 1$.

The details of this work are described in an article [2]. The source code and HTML documentation of the Agda formalization can be downloaded from `https://hott.github.io/M-types/`.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005.

[2] B. Ahrens, P. Capriotti, and R. Spadotti. Non-wellfounded trees in homotopy type theory. In *Proc. of TLCA '15*, *Leibniz Int. Proc. in Inform.*, Dagstuhl, to appear. arXiv:1504.0294

[3] T. Altenkirch, N. Ghani, P. Hancock, C. McBride, and P. Morris. Indexed containers. Journal version draft. `http://www.cs.nott.ac.uk/~txa/publ/jcont.pdf`.

[4] S. Awodey, N. Gambino, and K. Sojakova. Inductive types in homotopy type theory. In *Proc. of 27th Ann. IEEE Symp. on Logic in Comput. Sci., LICS '12*, pp. 95–104. IEEE CS, 2012.

[5] HoTT mailing list. Discussion on coinductive types on HoTT mailing list, `https://groups.google.com/d/msg/homotopytypetheory/tYRTcI2Opyo/PIrI6t5me-oJ`.

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

[7] B. van den Berg and F. De Marchi. Non-well-founded trees in categories. *Ann. of Pure and Appl. Logic*, 146(1):40–59, 2007.

Substitution Systems Revisited*

Benedikt Ahrens and Ralph Matthes

Institut de Recherche en Informatique de Toulouse, Université Paul Sabatier, France
{ahrens,matthes}@irit.fr

Abstract

Matthes and Uustalu (TCS 327(1–2):155–174, 2004) presented a categorical description of substitution systems capable of capturing syntax involving binding which is independent of whether the syntax is made up from least or greatest fixed points. We extend this work in two directions: we continue the analysis by creating more categorical structure, in particular by organizing substitution systems into a category and studying its properties, and we develop the proofs of the results of the cited paper and our new ones in UniMath, a recent library of univalent mathematics formalized in the Coq theorem prover.

In previous work, Matthes and Uustalu [6] define a notion of "heterogeneous substitution system", the purpose of which is to axiomatize substitution and its desired properties. Such a substitution system is given by an algebra of a signature functor, equipped with an operation—which is to be thought of as substitution—that is compatible with the algebra structure map in a suitable sense. The term "heterogeneous" refers to the fact that the underlying notion of signature encompasses variable binding constructions and also explicit substitution a. k. a. flattening. More precisely, the signature is based on a rank-2 functor H (an endofunctor on a category of endofunctors) for the respective domain-specific signature, to which a monadic unit is explicitly added. The latter corresponds to the inclusion of variables into the elements that are considered as terms (in a quite general sense) over their variable supplies. The name "rank-2 functor" stems from the rank of the type operator that transforms type transformations into type transformations—hence has kind $(\mathsf{Set} \to \mathsf{Set}) \to (\mathsf{Set} \to \mathsf{Set})$—which may be seen as backbone of H in case the base category is Set. In this rank-2 setting, the carrier of the algebra is an endofunctor, and since a monadic unit is already present, a natural question is if one obtains a monad. In that paper, it is then shown that for any heterogeneous substitution system this is indeed the case; multiplication of the monad is derived from the "substitution" operation which is parameterized by a morphism f of pointed endofunctors and consists in asking for a unique solution that makes a certain diagram commute. Monad multiplication and one of the monad laws is obtained from the existence of a solution in the case that f is the identity, while the other monad laws are derived from uniqueness for two other choices of f.

Furthermore, it is shown there that "substitution is for free" for both initial algebras as well as—maybe more surprisingly—for (the inverse of) final coalgebras: if the initial algebra, resp. terminal coalgebra, of a given signature functor exists, then it, resp. its inverse, can be augmented to a substitution system (for the former case, and in order to easily use generalized iteration [4], it is assumed that the functor $- \cdot Z$ has a right adjoint for every endofunctor Z). Indeed, it was one of the design goals of the axiomatic framework of heterogeneous substitution systems to be applicable to *non-wellfounded* syntax as well as to wellfounded syntax, whereas related work (e.g., [5, 2]) frequently only applies to wellfounded syntax.

Examples of substitution systems are thus given by the lambda calculus, with and without explicit flattening, but also by languages involving typing and *infinite* terms.

*The work of Benedikt Ahrens was partially supported by the CIMI (Centre International de Mathématiques et d'Informatique) Excellence program ANR-11-LABX-0040-CIMI within the program ANR-11-IDEX-0002-02 during a postdoctoral fellowship.

The goal of the present work is to extend and to formalize the work by Matthes and Uustalu [6]; in particular, we introduce a natural notion of *morphisms* of heterogeneous substitution systems, thus arranging them into a category. We then show that the construction of a monad from a heterogeneous substitution system from [6] extends functorially to morphisms.

Moreover, we prove that the substitution system obtained in [6] by equipping the initial algebra with a substitution operation, is initial in the corresponding category of substitution systems. This makes use of a general fusion law for generalized iteration [4]. However, we will show why a similar result cannot be expected for final coalgebras.

As an example of the usefulness of our results, we intend to express the resolution of explicit flattening of the lambda calculus as a(n initial) morphism of substitution systems.

We are in the process of formalizing our results in univalent foundations, more specifically basing ourselves on the UniMath library [1]. This basis of our formalization is suitable in that it provides extensionality (functional and propositional) in a natural way and hereby avoids the use of setoids that would otherwise be inevitable; indeed, since our results are not about categories *in abstracto* but use general categorical concepts in more concrete instances such as the endofunctor category over a given category or its extension by a "point", we need extensionality axioms for the instantiation. Functional extensionality is a consequence of the univalence axiom that we do not use directly in our formalization. We profit from the existing category theory library in UniMath. It is well-known that the handling of rank-2 functors needs universe polymorphism (otherwise, one would need a number of copies of the categorical concepts in the Coq vernacular, since, e. g., the endofunctor category has a higher universe level than its base category). In UniMath, this universe polymorphism is provided by switching off the control of universe levels (setting $\mathsf{Type} : \mathsf{Type}$). Still, this is not extensional type theory—there is a big difference between convertibility and equality. Already for the purpose of type-checking of statements taken from the paper version, we need to reformulate some laws to fit into the type-theoretic framework and to slightly reformulate the statements (in concrete instances, the differences would plainly disappear). The culprit here is that convertibility of types of natural transformations is not preserved under application of an abstract rank-2 functor H.

At the time of writing this abstract, we have not yet finished the formalization; what we do have is a full formalization of the main result of Matthes and Uustalu [6, Theorem 10] that yields a monad for every heterogeneous substitution system, and moreover our refinement stating that this gives rise to a faithful functor into the category of monads. The formalization can be consulted at [3]. We hope that we will have formalized the other new theoretical results before presenting this work at the TYPES conference.

References

[1] UniMath. `http://unimath.org`.

[2] B. Ahrens. Extended initiality for typed abstract syntax. *Log. Methods in Comput. Sci.*, 8(2:1), 2012.

[3] B. Ahrens and R. Matthes. Substitution systems revisited: formalization in Coq. `https://github.com/benediktahrens/substitution`.

[4] R. S. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Asp. Comput.*, 11(2):200–222, 1999.

[5] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of 14th Ann. IEEE Symp. on Logic in Computer Science, LICS '99*, pp. 193–202. IEEE CS, 1999.

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

Towards a Theory of Higher Inductive Types

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

[1] School of Computer Science, University of Nottingham, United Kingdom
[2] Dept. of Computer and Inform. Sci., University of Strathclyde, United Kingdom
{txa|pvc|gxd}@cs.nott.ac.uk, fredrik.nordvall-forsberg@strath.ac.uk

Homotopy Type Theory extends intensional Martin-Löf Type Theory with two new features: Voevodsky's Univalence Axiom and *higher inductive types* (HITs). Both are motivated by interpretations of Type Theory into abstract homotopy theory, where intuitively types are interpreted as topological spaces, terms are interpreted as points, and the identity type is interpreted as the space of paths between points. Specific higher inductive types have successfully been deployed, e.g. for synthetic homotopy theory (e.g. [6, 8]) and computer science applications [4], but a general theory of higher inductive types is so far lacking, although partial results have been achieved by Sojakova [10] for the restricted class of W-*suspensions*. Shulman and Lumsdaine [9] also consider a semantics for an unspecified collection of HITs in model categories.

Our first goal is to define a general class of HITs, together with their introduction and elimination rules, in such a way that Shulman and Lumsdaine's semantics applies. We want our notion of HITs to include naturally occurring examples such as the following definition of the torus T^2:

```
data T² : Type where
    b :  T²
    v :  b = b
    h :  b = b
 surf :  v · h = h · v
```

Here, we say that T^2 is the least type with a point `b`, two paths `v` and `h` from `b` to `b`, and a 2-dimensional path (i.e., a surface) `surf` from the composite path `v · h` to the composite path `h · v`. Current treatments of HITs, such as W-suspensions, or the treatment in the Agda HoTT library [7] cannot handle T^2 because of the 2-dimensional constructor `surf` (the *hub-and-spoke* construction can reduce HITs with higher-dimensional constructors to HITs with at most 1-dimensional constructors, but this is something we want to prove using our framework, not assume!). Note also that `surf` presents an additional difficulty in that its target `v · h = h · v` is a path between two expressions constructed from previous constructors.

In general, a constructor c_k for a HIT $A : \mathsf{Type}$ has type

$$\mathsf{c}_k : (x : F_k(A, \mathsf{c}_0, \ldots, \mathsf{c}_{k-1})) \to G_k(A, \mathsf{c}_0, \ldots, \mathsf{c}_{k-1}, x) \tag{1}$$

where F_k and G_k are functors (with G_k always returning an iterated path space), and c_i are the previously given constructors. For the torus T^2, we have $F_0(A) = \mathbf{1}$, $F_1(A, c_0) = \mathbf{1}$, $F_2(A, c_0, c_1) = \mathbf{1}$ and $F_3(A, c_0, c_1, c_2) = \mathbf{1}$, and $G_0(A, x) = A$, $G_1(A, c_0, x) = (c_0 = c_0)$, $G_2(A, c_0, c_1, x) = (c_0 = c_0)$ and $G_3(A, c_0, c_1, c_2, x) = c_1 \cdot c_2 =_{c_0 = c_0} c_2 \cdot c_1$. Thus, we see that $F_0 : \mathsf{Type} \to \mathsf{Type}$, and $F_{k+1} : (F_k, G_k)\text{-}\mathsf{Alg} \to \mathsf{Type}$, where (F_k, G_k)-Alg is the category of "algebras" (A, c_k) with c_k as in (1).

For this to give rise to meaningful definitions, we need restrictions on F_k and G_k. Intuitively, we need F_k to be strictly positive, and G_k to be a path space where the left and right hand side of the equation is natural in the input. Ordinary endofunctors are strictly positive exactly

when they can be represented by *containers* [1]. Since we are dealing with non-endofunctors F_k, we need to generalise this concept to that of familially representable functors [5]: a functor $H : \mathbb{C} \to \mathsf{Type}$ is familially representable, or a $\mathbb{C}$-*container*, if there exists a type S and a family $P : S \to |\mathbb{C}|$ such that $H(X) \cong \Sigma\, s : S\,.\,\mathsf{Hom}_{\mathbb{C}}(P(s), X)$. Ordinary containers are exactly Type-containers. By constructing a left adjoint $L_k : \mathsf{Type} \to (F_k, G_k)\text{-}\mathsf{Alg}$ to the forgetful functor $U_k : (F_k, G_k)\text{-}\mathsf{Alg} \to \mathsf{Type}$ (with $L(A) =$ "the free algebra on A"), we can show that all G_k are $(F_{k-1}, G_{k-1})\text{-}\mathsf{Alg}$-containers if all F_k are. Since $\mathbb{C}$-containers come with a notion of container morphisms, representing natural transformations between functors, and all functors in sight are containers, we can thus ask for container morphisms to represent G_k. This guarantees a meaningful notion of general HITs with arbitrarily complicated constructors.

Making all of these definitions precise, however, is not a trivial task. In fact, the types involved do not form proper categories in the sense of Ahrens et al. [2]: not even the base "category" Type. Since we make heavy use of containers and container morphisms, which can be defined only in terms of objects, it might seem that it should be possible to get away with a more naive definition of "category" without the restriction that the morphisms form a set. Unfortunately, this approach is bound to fail, since the number of coherence properties that we need to impose turns out to equal the number of constructors. That is, even though a simple definition of "category" including objects, morphisms, composition and associativity is enough to express inductive specifications with 2 constructors, at step 3 we would not be able to define algebra morphisms.

One solution we are currently investigating [3] is to modify the type theory by adding a *strict equality* type constructor, to get something very similar to Voevodsky's proposed *Homotopy Type System* [11], but without the reflection rule for strict equality. In this system, one can define *strict categories* and *strict functors*, whose laws are expressed using strict equality. With this definition, Type can be regarded as a strict category, and containers give rise to strict functors.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Representing nested inductive types using W-types. In *Proc. of ICALP 2004*, v. 3142 of *Lect. Notes in Comput. Sci.*, pp. 59–71. Springer, 2004.

[2] B. Ahrens, K. Kapulkin, and M. Shulman. Univalent categories and the Rezk completion. *Math. Struct. in Comp. Science*, 25(5):1010–1039, 2015.

[3] T. Altenkirch. The coherence problem in HoTT. Talk at the FP Lab Away Day, July 2014. `http://www.cs.nott.ac.uk/~txa/talks/away14.pdf`.

[4] C. Angiuli, E. Morehouse, D. R. Licata, and R. Harper. Homotopical patch theory. In *Proc. of ICFP '14*, pp. 243–256. ACM, 2014.

[5] A. Carboni and P. Johnstone. Connected limits, familial representability and Artin glueing. *Math. Struct. in Comput. Sci.*, 5(4):441–459, 1995.

[6] E. Cavallo. The Mayor-Vietoris sequence in HoTT. Talk at Oxford Workshop on Homotopy Type Theory, Nov. 2014. `http://www.qmac.ox.ac.uk/events/Talkslides/EvanCavallo.pdf`.

[7] HoTT-Agda. Code repository, 2015. `https://github.com/hott/hott-agda/`.

[8] D. R. Licata and E. Finster. Eilenberg-Maclane spaces in Homotopy Type Theory. In *Proc. of CSL-LICS '14*, art. 66. ACM, 2014.

[9] M. Shulman and P. L. Lumsdaine. Semantics of higher inductive types. `http://ncatlab.org/homotopytypetheory/files/hit-semantics.pdf`, 2013.

[10] K. Sojakova. Higher inductive types as homotopy-initial algebras. In *Proc. of POPL '15*, pp. 31–42. ACM, 2015.

[11] V. Voevodsky. A type system with two kinds of identity types. `https://uf-ias-2012.wikispaces.com/file/view/HTS_slides.pdf/410105196/HTS_slides.pdf`, 2013.

Infinite Structures in Type Theory: Problems and Approaches

Thorsten Altenkirch, Paolo Capriotti and Nicolai Kraus

School of Computer Science, University of Nottingham, United Kingdom
{txa|pvc|ngk}@cs.nott.ac.uk

When exploring the possibilities of Homotopy Type Theory (HoTT), one quickly realises that many constructions come with coherence problems. This is already apparent when we try to define the concept of *monoid* in full generality: we begin with a type A of elements and a composition function $\circ$. Then we add the requirement that $\circ$ is associative, which can be expressed using the identity type. However, in many situations, associativity might not be enough: we need a form of coherence for the equalities produced by it (the "pentagon"). But these coherence proofs may be required to satisfy their own coherence condition, and this process continues indefinitely, leading to a tower of "coherence properties" which we currently cannot capture with a single definition within type theory. A possible solution is to restrict ourselves to types of some given (small) truncation level, and in the described example, it is justifiable that A should be a *set* (in the sense of HoTT). However, there are a number of interesting cases where either this is not possible or makes no sense. We can think of the following:

1. Defining semi-simplicial types (see e.g. [2]). More generally, representing Reedy fibrant diagrams [7] internally.
2. Given a powerful enough mutual induction principle for higher inductive types, it is possible to define the syntax of HoTT in itself. However, any attempts to define an interpretation function that maps syntactic types to outer types is stymied by the appearance of coherence problems, when mapping definitional equalities of syntactical elements to propositional equalities. A discussion can be found in [6].
3. The current formulation of category theory in HoTT assumes 0-truncated hom-sets [1]. This has the problem of excluding the "category" of types and functions, and many other interesting categorical structures, for which we would need to internalise a notion of $(\infty, 1)$-category.
4. If we want to construct functions $\|A\| \to B$, previous work [4] shows that we have (depending on the truncation level of B) to provide a tower of coherence conditions. In the current theory, we can only express finite parts, and we would like to be able to internalise and formalise the complete result.

All these questions are related and we expect that none of them is solvable in the theory that is considered in the standard reference [8]. Let us elaborate on the first, which we can express as follows: given a type expression E, is it possible to define a family of types $M : \mathbb{N} \to \mathsf{Type}$ such

that E can be given the type $(n : \mathbb{N}) \to M_n \to \mathsf{Type}$ and M satisfies[1]

$$\begin{aligned}
M_0 &\cong \mathsf{1} \\
M_1 &\cong E_0 \times E_0 \\
M_2 &\cong (x_0, x_1, x_2 : E_0) \times E_1(x_0, x_1) \times E_1(x_0, x_2) \times E_1(x_1, x_2) \\
M_3 &\cong (x_0, x_1, x_2, x_3 : E_0) \times (x_{01} : E_1(x_0, x_1)) \times (x_{02} : E_1(x_0, x_2)) \times (x_{03} : E_1(x_0, x_3)) \\
&\quad \times (x_{12} : E_1(x_1, x_2)) \times (x_{13} : E_1(x_1, x_3)) \times (x_{23} : E_1(x_2, x_3)) \\
&\quad \times E_2(x_0, x_1, x_2, x_{01}, x_{02}, x_{12}) \times E_2(x_0, x_1, x_3, x_{01}, x_{03}, x_{13}) \\
&\quad \times E_2(x_0, x_2, x_3, x_{02}, x_{03}, x_{23}) \times E_2(x_1, x_2, x_3, x_{12}, x_{13}, x_{23})
\end{aligned} \tag{1}$$

and so on? At first sight, it looks as if M should be relatively straightforward to define by induction on its index. However, a lot of time and effort has already been spent on this (see, for example, Herbelin's exposition [2] and Shulman's discussion [6]), and it seems that none of the proposed constructions work. It is not easy to pinpoint what goes wrong.

We could try the following: first, we define the category Δ_+ in type theory. It has nonempty finite sets as objects and strictly increasing functions as morphisms.Then, we can attempt to define a double-indexed family $M_n^{(k)}$ by induction on k, such that the sequence $M_n^{(k)}$ stabilises for $k \geq n$, and such that $M_n^{(k)}$ is a Δ_+-functor in n. Our intention is then to take $M_n :\equiv M_n^{(n)}$. We set $M_n^{(0)} :\equiv \mathsf{1}$. Then, inductively, we define

$$M_n^{(k+1)} :\equiv (x : M_n^{(k)}) \times ((\sigma : \Delta_+(k, n)) \to E_k(\sigma^\star x)), \tag{2}$$

where $\sigma^\star$ is the functorial action of $M^{(k)}$ (defined at the same time as $M_n^{(k)}$). Unfortunately, this does not type-check. If we (manually or by a script) start to write down the sequence $M_0, M_1, M_2, \ldots$, together with appropriate E_i, following the formula (2), then this does type-check.[2] However, as soon as the indices are variables of type $\mathbb{N}$ (instead of fixed numerals), the construction does not type-check because the functorial action of $M^{(k)}$ is not strict. This suggests that a judgmental η-law for natural numbers, or some form of equality reflection, would provide a solution. A similar observation, we believe, motivated Voevodsky to start the development of HTS.

The problem of constructing "Reedy-fibrant diagrams", or "infinite Σ-types", has been discussed in the community quite frequently (see, for example, Shulman's blog post and its discussion [6]), and several people have argued that it would be desirable to formulate a version of the HoTT theory in which this is possible. Motivations that are similar to ours have led to the development of Voevodsky's HTS [9], a 2-level system, and also to Hickey's *very dependent types* [3] that have been suggested in the context of NuPRL (and for which it is unclear whether they can be made sense of in HoTT).

Attempting to provide an extension of the "standard theory" to perform such constructions, we suggest a system with two different equalities. This is the same approach that is taken in the development of HTS. However, we hope to be able to avoid resorting to the reflection rule for strict equality, which makes typechecking undecidable. Currently, we believe that this is possible in such a way that we can use Agda to provide a nice implementation of our system.

[1] We write $(a : A) \times B(a)$ for $\Sigma(a : A).B(a)$ and (later) $(a : A) \to B(a)$ for $\Pi(a : A).B(a)$.

[2] An interesting observation is that, if the type theory has η for both Π- and Σ-types (as in Agda, but not in Coq), Δ_+ can be implemented in such a way that all categorical laws hold strictly [5]. This is a requirement for the claim that $M_1, M_2, \ldots$ type-check. It looks as if this strictness could be useful for the construction in general, but so far, we are not sure whether it is indeed relevant.

References

[1] B. Ahrens, K. Kapulkin, and M. Shulman. Univalent categories and the Rezk completion. *Math. Struct. in Comput. Sci.*, 25(5):1010–1039, 2015.

[2] H. Herbelin. A dependently-typed construction of semi-simplicial types. *Math. Struct. in Comput. Sci.*, 25(5):1116–1131, 2015. 2015.

[3] J. J. Hickey. Formal objects in type theory using very dependent types. In *Informal Proc. of Foundations of Object Oriented Languages 3*, 1996.

[4] N. Kraus. The general universal property of the propositional truncation. In *Proc. of TYPES '14*, *Leibniz Int. Proc. in Inform.*, Dagstuhl, to appear. arXiv:1411.2682

[5] N. Kraus. *Truncation Levels in Homotopy Type Theory.* PhD thesis, University of Nottingham, 2015.

[6] M. Shulman. Homotopy type theory should eat itself (but so far, it's too big to swallow). Blog post at homotopytypetheory.org, 3 March 2014.

[7] M. Shulman. Univalence for inverse diagrams and homotopy canonicity. *Math. Struct. in Comput. Sci.*, 25(5):1203–1277, 2015.

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

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

A Nominal Syntax for Internal Parametricity

Thorsten Altenkirch and Ambrus Kaposi

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

Abstract

We define a type theory with internal parametricity together with an operational semantics. This is a first step towards the goal of defining cubical type theory where univalence is an admissible rule.

1 Introduction

Parametricity [6] is the principle that terms respect logical relations. Univalence [5] can be understood as a principle that terms respect extensional equality.

Two pairs are logically related if they are componentwise related. Two functions are logically related if they map related inputs to related outputs. Two types are logically related if there is any relation between them.

Two pairs are equal if they are componentwise equal. Two functions are equal if they map equal inputs to equal outputs. Two types are equal if there is a relation between them which is a graph of an equivalence.

The above correspondence suggests that if we have a type theory with internal parametricity, by replacing the definition of "relatedness" for the universe and adjusting the rest of theory, we get a theory with univalence. The current formulations of Homotopy Type Theory (eg. [5]) lack a computational understanding of univalence. Internal parametricity was given computational meaning by [3] and [2]. Our work can be seen as a reformulation of [3] in an explicit substitution calculus using dimension names instead of permutations which simplifies the presentation. One difference from [2] is that we don't decorate typing judgements with a list of dimensions.

We first extend the syntax of type theory with the metatheoretic proof that every term is parametric (section 2). However, this is not enough for internal parametricity, since the function which maps a term to its parametricity proof cannot be typed, because the domain and codomain live in different contexts. To solve this problem, we introduce a substitution from a context to the extended context of related elements (section 3). Finally, we discuss how to extend the theory to use equivalence instead of any relation for "relatedness" (section 4).

2 External parametricity

Parametricity says that logically related interpretations of a term are logically related. More precisely, given $\Gamma \vdash t : A$, $\rho, \rho' : \emptyset \Rightarrow \Gamma$ and a witness that ρ and ρ' are related in the logical relation generated by Γ, then $t[\rho]$ and $t[\rho']$ will be related in the logical relation generated by A. We formalize this by the following rule:

$$\frac{\Gamma \vdash t : A}{\Gamma^i \vdash t^i : t[\mathsf{0}_i\Gamma] \sim_{A^i} t[\mathsf{1}_i\Gamma]}$$

Γ^i contains two copies of Γ (ρ and ρ' above) and a logical relation between them. $\sim_{A^i}$ is the logical relation generated by A. It relates the two interpretations of the term t, in the two

different copies of Γ, which are projected out by the substitutions $\mathsf{0}_{i\Gamma}$, $\mathsf{1}_{i\Gamma}$. t^i is the witness of this relation.

The context Γ^i contains two copies of Γ and witnesses that they are pointwise related:

$$(\Gamma.x : A)^i \equiv \Gamma^i.x_{i0} : A[\mathsf{0}_{i\Gamma}].x_{i1} : A[\mathsf{1}_{i\Gamma}].x_{i2} : x_{i0} \sim_{A^i} x_{i1}).$$

The operation $-^i$ on terms is defined by induction on the term structure, eg. for the universe:

$$A \sim_{\mathsf{U}^i} B \equiv A \to B \to \mathsf{U}.$$

3 Internal parametricity

To internally derive that every function of type $\Pi(A : \mathsf{U}).A \to A$ is identity, we would need a term t expressing parametricity for f assuming f:

$$\begin{aligned} f : \Pi(A : \mathsf{U}).A \to A \vdash t : &\Pi(A_{i0}, A_{i1} : \mathsf{U}, A_{i2} : A_{i0} \to A_{i1} \to \mathsf{U})\,. \\ &\Pi(x_{i0} : A_{i0}, x_{i1} : A_{i1}, x_{i2} : A_{i2}\, x_{i0}\, x_{i1})\,.\, A_{i2}\,(f\, A_{i0}\, x_{i0})\,(f\, A_{i1}\, x_{i1}). \end{aligned}$$

We could choose t to be f^i, but that lives in the context $(f : \Pi(A : \mathsf{U}).A \to A)^i$.

This motivates the definition of a substitution that takes us into the $-^i$-d context. We define this substitution mutually with the term former refl_i. The typing rules are as follows:

$$\frac{\Gamma \vdash}{\mathsf{R}_{i\Gamma} : \Gamma \Rightarrow \Gamma^i} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{refl}_i\, t : t \sim_{A^i[\mathsf{R}_{i\Gamma}]} t}$$

Now the above term t can be defined as $f^i[\mathsf{R}_i]$.

We give an operational semantics for the type theory which has the substitution R.

4 Equivalence

If we replace the definition of U^i by equivalence, i.e. $A \sim_{\mathsf{U}^i} B \equiv \Sigma(\sim : A \to B \to \mathsf{U}).\big(\Pi(x : A).\mathsf{isContr}(\Sigma(y : B).x \sim y)\big) \times \big(\Pi(y : B).\mathsf{isContr}(\Sigma(x : A).x \sim y)\big)$, we are forced to add Kan fillers for every type, as in [4]. This is ongoing work [1].

References

[1] T. Altenkirch and A. Kaposi. A syntax for cubical type theory. Unpublished draft, 2014.

[2] J.-P. Bernardy and G. Moulin. Type-theory in color. In *Proc. of 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '13*, pp. 61–72. ACM, 2013.

[3] J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *Proc. of 27th Ann. IEEE/ACM Symp. on Logic in Computer Science, LICS '12*, pp. 135–144. IEEE CS, 2012.

[4] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In R. Matthes and A. Schubert, eds., *Proc. of 19th Int. Conf. on Types for Proofs and Programs, TYPES 2013*, v. 26 of *Leibniz Int. Proc. in Inform.*, pp. 107–128. Dagstuhl, 2014.

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

[6] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, ed., *Proc. of IFIP 9th World Computer Congress, Information Processing '83*, pp. 513–523. North-Holland, 1983.

Dependent Inductive and Coinductive Types through Dialgebras in Fibrations

Henning Basold[1,2] and Herman Geuvers[1,3]

[1] Institute for Comput. and Inform. Sciences, Radboud Universiteit, The Netherlands
[2] CWI, The Netherlands
[3] Technische Universiteit Eindhoven, The Netherlands
{h.basold|h.geuvers}@cs.ru.nl

It is well-known that dependent type theories with fixed type constructors can be interpreted over certain fibrations, see, for example, [2]. On the other hand, Hagino [1] showed how a language with constructors for inductive and coinductive types *without* term dependencies can be treated uniformly through the use of special dialgebras. However, at least to our knowledge, these techniques have not been combined, yet. Hence, the goal of the present work is to interpret dependent inductive and coinductive data types using certain dialgebras in fibrations.

The reason for the choice of dialgebras is that we can interpret data types without having to require the existence of (co)products separately, see below. An example of such a data type are vectors, that is, lists combined with their length, which could be declared in Agda by

$$\begin{aligned}&\textbf{data}\ \mathrm{Vec}\ (\mathrm{A} : \mathbf{Set}) : \mathbb{N} \to \mathbf{Set}\ \textbf{where}\\ &\quad \mathrm{nil}\ :\ \mathbf{1} \to \mathrm{Vec\ A\ 0}\\ &\quad \mathrm{cons}\ :\ (\mathrm{k}\ :\ \mathbb{N}) \to \mathrm{A} \times \mathrm{Vec\ A\ k} \to \mathrm{Vec\ A\ (k + 1)}\end{aligned}$$

where $\mathbf{1}$ is the one-element type and $\mathbb{N}$ the natural numbers. The important observation is that Vec has two constructors, nil and cons, with the following properties

1. nil has no *local dependencies*, whereas cons introduces locally a fresh variable $k : \mathbb{N}$.
2. nil has a trivial argument of type $\mathbf{1}$, whereas cons has as argument an element of A and an element of $(\mathrm{Vec}\, A\, n)[k/n]$ in context $k : \mathbb{N}$.
3. nil constructs an element of $(\mathrm{Vec}\, A\, n)[0/n]$, whereas cons constructs an element of the type $(\mathrm{Vec}\, A\, n)[k + 1/n]$ in context $k : \mathbb{N}$.

These properties allow us to interpret the constructors as a dialgebra.

The semantics of these constructors can be explained in the category $\mathbf{Set}^{\mathbb{N}}$. Given a set I, the category $\mathbf{Set}^I$ has set families $X = \{X_i\}_{i\in I}$ as objects and families $\{f_i : X_i \to Y_i\}_{i\in I}$ of functions as morphisms $f : X \to Y$. Moreover, for every function $g : I \to J$ there is a functor $g^* : \mathbf{Set}^J \to \mathbf{Set}^I$, the *reindexing* or *substitution* along g, given by $g^*(X) = \{X_{g(i)}\}_{i\in I}$. The categories $\mathbf{Set}^I$ and the reindexing functors can be organised into the families fibration $\mathrm{Fam}(\mathbf{Set})$, see for example [2].

The example of vectors is represented in $\mathrm{Fam}(\mathbf{Set})$ as follows. We define the family $\mathrm{Vec}\, A = \{A^n\}_{n\in\mathbb{N}}$, which lives in $\mathbf{Set}^{\mathbb{N}}$. The constructors, however, live in different categories, namely nil is given by the singleton family $\{\mathrm{nil}_* : \mathbf{1} \to \mathrm{Vec}\, A\, 0\}_{*\in\mathbf{1}}$ in $\mathbf{Set}^{\mathbf{1}}$, and the constructor cons is given by the morphism $\mathrm{cons} = \{\mathrm{cons}_k : A \times \mathrm{Vec}\, A\, k \to \mathrm{Vec}\, A\, (k + 1)\}_{k\in\mathbb{N}}$ in $\mathbf{Set}^{\mathbb{N}}$. We can see these constructors as one morphism $(\mathrm{nil}, \mathrm{cons})$ in the product category $\mathbf{Set}^{\mathbf{1}} \times \mathbf{Set}^{\mathbb{N}}$.

To understand the nature of these constructors, we draw on dialgebras, as pioneered by Hagino [1]. We define two functors $F, G : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}} \times \mathbf{Set}^{\mathbb{N}}$, such that F captures the domain and G the codomain of the constructors:

$$F(X) = (\{\mathbf{1}\}, \{A \times X_k\}_{k\in\mathbb{N}}) \qquad G(X) = (\{X_0\}, \{X_{k+1}\}_{k\in\mathbb{N}})$$

with the obvious action on morphisms. The constructors for Vec form then an (F, G)-dialgebra, that is, $(\mathrm{nil}, \mathrm{cons}) : F(\mathrm{Vec}\, A) \to G(\mathrm{Vec}\, A)$. In fact, $(\mathrm{nil}, \mathrm{cons})$ is an *initial* (F, G)-dialgebra.

The final step is to note the special shape of G. On the natural numbers, we have the two maps $z : \mathbf{1} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$ given by $z(*) = 0$ and $s(n) = n + 1$, respectively. From these we get the two reindexing functors $z^* : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}}$ and $s^* : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbb{N}}$, which we can combine, using pairing of functors, into $\langle z^*, s^* \rangle$. Looking closely at the definitions, we find that this is exactly G.

Using this analysis, we can define what (non-nested) data types in Fam(**Set**) are. We call a pair of functors (F, G) with $F, G : \mathbf{Set}^I \to \prod_{i=1,\dots,n} \mathbf{Set}^{J_i}$ and $G = \langle f_1^*, \dots, f_n^* \rangle$ a *data type* signature. An *inductive* data type is then an initial (F, G)-dialgebra and a *coinductive* data type is a final (G, F)-dialgebra (note the order).

An example of a coinductive data type are partial streams, which are streams indexed by their (potentially infinite) length. These are given by the following pseudo-Agda declaration.

$$\begin{aligned} &\textbf{codata}\ \mathrm{PStr}\ (\mathrm{A} : \mathbf{Set}) : \mathbb{N}^\infty \to \mathbf{Set}\ \textbf{where} \\ &\quad \mathrm{hd} : (\mathrm{n} : \mathbb{N}^\infty) \to \mathrm{PStr}\ \mathrm{A}\ (s_\infty\ \mathrm{n}) \to \mathrm{A} \\ &\quad \mathrm{tl} : (\mathrm{n} : \mathbb{N}^\infty) \to \mathrm{PStr}\ \mathrm{A}\ (s_\infty\ \mathrm{n}) \to \mathrm{PStr}\ \mathrm{A}\ \mathrm{n} \end{aligned}$$

where $\mathbb{N}^\infty$ is the coinductive type of natural numbers extend by infinity and $s_\infty : \mathbb{N}^\infty \to \mathbb{N}^\infty$ the definable successor. The type PStr has two *destructors*, which can only be applied to partial streams that contain at least one element. They are the final $(\langle s_\infty^*, s_\infty^* \rangle, H)$-dialgebra, where $H : \mathbf{Set}^{\mathbb{N}^\infty} \to \mathbf{Set}^{\mathbb{N}^\infty} \times \mathbf{Set}^{\mathbb{N}^\infty}$ is given by $H(X) = (w(A), X)$ and $w : \mathbf{Set}^{\mathbf{1}} \to \mathbf{Set}^{\mathbb{N}^\infty}$ is the weakening functor given by reindexing along $!_{\mathbb{N}^\infty} : \mathbb{N}^\infty \to \mathbf{1}$.

It is noteworthy that coinductive data types are perfectly dual to inductive types. They are also an extension of the syntax for record types in Agda, in the sense that we are allowed to refine the domain of destructors using substitutions.

Another interesting example are products along arbitrary maps. In our pseudo-Agda notation, these are given by

$$\begin{aligned} &\textbf{codata}\ \textstyle\prod_{\_}\ (\mathrm{f} : \mathrm{I} \to \mathrm{J})\ (\mathrm{B} : \mathrm{I} \to \mathbf{Set}) : \mathrm{J} \to \mathbf{Set}\ \textbf{where} \\ &\quad \mathrm{app} : (\mathrm{i} : \mathrm{I}) \to \textstyle\prod_f\ \mathrm{B}\ (\mathrm{f\ i}) \to \mathrm{B\ i} \end{aligned}$$

It turns out that, when interpreted in Fam(**Set**), $\prod_f$ is right-adjoint to f^*, which is the usual definition of the product. Coproducts (Σ-types) can be defined analogously as inductive types. Note though that the $\Sigma x : A.B$ is usually given in Agda as record-type with projections on A and B, which gives a strong sum elimination that we do not have in the present setting. We will discuss this in conjunction with induction.

The talk will consist of the following topics. We will generalise the idea of dependent inductive and coinductive data types, we outlined above, to allow for nested data types. Moreover, we will see how these data types relate to algebras and coalgebras, and we will discuss induction and coinduction in this setting. Finally, if time permits, and we will investigate a Beck-Chevalley condition for data types.

This work has been submitted to CALCO 2015.

References

[1] T. Hagino. A typed lambda calculus with categorical type constructors. In *Proc. of 2nd Conf. on Category Theory in Computer Science, CTCS '87*, v. 283 of *Lect. Notes in Comput. Sci.*, pp. 140–157. Springer, 1987.

[2] B. Jacobs. *Categorical Logic and Type Theory.* v. 141 in *Studies in Logic and the Foundations of Mathematics.* North Holland, 1999.

Dialgebra-Inspired Syntax for Dependent Inductive and Coinductive Types

Henning Basold[1,2] and Herman Geuvers[1,3]

[1] Institute for Comput. and Inform. Sciences, Radboud Universiteit, The Netherlands
{h.basold|h.geuvers}@cs.ru.nl
[2] CWI, The Netherlands
[3] Technische Universiteit Eindhoven, The Netherlands

The goal of this work is the development of a syntax that treats inductive and coinductive types with term dependencies on a par. We achieve this by extending Hagino's idea [2] to dependent types, in that we interpret data types as dialgebras in fibrations. This follows the ideas we present in the talk "Dependent Inductive and Coinductive Types Through Dialgebras in Fibrations". We note that our syntax for inductive types is similar to GADTs [3], and that coinductive types are defined by their destructors like in [1].

The context, types and terms of our calculus are introduced using the following judgements.v

- $\vdash \Delta \mid \Gamma$ **ctx** – The type variable context Δ and term variable context Γ are well-formed.
- $\Delta \mid \Gamma \vdash A : *$ – The type A is well-formed in the context $\Delta \mid \Gamma$.
- $\Gamma \vdash t : A$ – The term t is well-formed and of type A in the context Γ.
- $f : \Gamma_1 \to \Gamma_2$ – The context morphism f is well-formed with type $\Gamma_1 \to \Gamma_2$.

The context judgement is given by the following rules.

$$\frac{}{\vdash \emptyset \mid \emptyset \ \mathbf{ctx}} \qquad \frac{\emptyset \mid \Gamma \vdash A : *}{\vdash \Delta \mid \Gamma, x : A \ \mathbf{ctx}} \qquad \frac{\vdash \Delta \mid \Gamma' \ \mathbf{ctx} \qquad \vdash \Delta \mid \Gamma \ \mathbf{ctx}}{\vdash \Delta, (\Gamma' \vdash X : *) \mid \Gamma \ \mathbf{ctx}}$$

It is important to note that, whenever a term variable is introduced into a context, its type is not allowed to use free type variables. This ensures that types are strictly positive.

The notion of substitution, we use, builds on *context morphisms* that are formed as follows.

$$\frac{\vdash \Delta \mid \Gamma \ \mathbf{ctx}}{() : \Gamma \to \emptyset} \qquad \frac{f : \Gamma_1 \to \Gamma_2 \qquad \Gamma_1 \vdash t : A[f]}{(f, t) : \Gamma_1 \to (\Gamma_2, x : A)}$$

Explicitly, if $\Gamma_2 = x_1 : A_1, \ldots, x_n : A_n$, then a context morphism $f : \Gamma_1 \to \Gamma_2$ is a tuple $f = (f_1, \ldots, f_n)$ with $\Gamma_1 \vdash f_i : A_i[(f_1, \ldots, f_{i-1})]$ for each $i = 1, \ldots, n$.

The judgement for type construction is given, together with the usual contraction, weakening and exchange rules for type variables, by the following rules.

$$\frac{}{\Delta, (\Gamma' \vdash X : *) \mid \Gamma, \Gamma' \vdash X : U_i} \ \textbf{(TyVar-I)} \qquad \frac{\Delta \mid \Gamma_1 \vdash A : * \qquad g : \Gamma_2 \to \Gamma_1}{\Delta \mid \Gamma_2 \vdash A[g] : *} \ \textbf{(Subst-Ty)}$$

$$\frac{\Delta, (\Gamma \vdash X : *) \mid \Gamma_k \vdash A_k : * \qquad f_k : \Gamma_k \to \Gamma \qquad k = 1, \ldots, n \qquad \rho \in \{\mu, \nu\}}{\Delta \mid \Gamma \vdash \rho(X; \vec{f}; \vec{A}) : *} \ \textbf{(FP-Ty)}$$

Note that substitutions with context morphisms are part of the syntax and that type variables come with a term variable context. These contexts determine the context in which an initial/-final dialgebra lives. The dialgebras essentially bundle the *local* context, the domain and the codomain of their constructors respectively destructors, as we will see.

This brings us to the rules for term constructions, the last judgement we have to define. The corresponding rules use substitutions of a type B for a variable X in a type A, denoted by $A\{BX\}$, which is defined as expected. All rules involving initial or final dialgebras have as side-condition that the corresponding types are well-formed.

$$\frac{\Delta \mid \Gamma \vdash A : *}{\Delta \mid \Gamma, x : A \vdash x : A}\ \textbf{(Proj)} \qquad \frac{\Delta \mid \Gamma_1 \vdash t : A \qquad g : \Gamma_2 \to \Gamma_1}{\Delta \mid \Gamma_2 \vdash t[g] : A[g]}\ \textbf{(Subst)}$$

$$\frac{}{\Gamma_k, x : A_k\{\mu(X; \overrightarrow{f}; \overrightarrow{A})X\} \vdash \alpha_k : \mu(X; \overrightarrow{f}; \overrightarrow{A})[f_k]}\ \textbf{(Ind-I)}$$

$$\frac{}{\Gamma_k, x : \nu(X; \overrightarrow{f}; \overrightarrow{A})[f_k] \vdash \xi_k : A_k\{\nu(X; \overrightarrow{f}; \overrightarrow{A})X\}}\ \textbf{(Coind-E)}$$

$$\frac{\Gamma \vdash C : * \qquad \Gamma_k, y : A_k\{CX\} \vdash g_k : C[f_k]}{\Gamma, z : \mu(X; \overrightarrow{f}; \overrightarrow{A}) \vdash \mathrm{rec}\ \overrightarrow{g} : C}\ \textbf{(Ind-E)}$$

$$\frac{\Gamma \vdash C : * \qquad \Gamma_k, y : C[f_k] \vdash g_k : A_k\{CX\}}{\Gamma, z : C \vdash \mathrm{corec}\ \overrightarrow{g} : \nu(X; \overrightarrow{f}; \overrightarrow{A})}\ \textbf{(Coind-I)}$$

We see that the domain of the constructors α_k for $\mu(X; \overrightarrow{f}; \overrightarrow{A})$ is determined by A_k and their codomain by substituting along f_k. Dually, the domain of destructors ξ_k is given by substituting f_k and their codomain by A_k. Note also that the bound variables in (co)recursions are implicit.

This concludes the definition of our proposed calculus. Note that there are no primitive type constructors for $\to$-, Π- or Σ-types, all of these are, together with the corresponding (weak) introductions and eliminations, definable in the above calculus. We will see this in the talk.

As basic example, assuming we have already defined the singleton type $\mathbf{1}$ and binary products, then we can define vectors $\Gamma \vdash \mathrm{Vec}\, A : *$ in context $\Gamma = n : \mathbb{N}$ by

$$\begin{array}{ll} \mathrm{Vec}\, A = \mu(X; (f_1, f_2); (\mathbf{1}, A \times X[k])) & \\ \Gamma_1 = \emptyset & \Gamma_2 = k : \mathbb{N} \\ f_1 = (0) : \Gamma_1 \to \Gamma & (\Gamma \vdash X : *) \mid \Gamma_1 \vdash \mathbf{1} : * \\ f_2 = (k+1) : \Gamma_2 \to \Gamma & (\Gamma \vdash X : *) \mid \Gamma_2 \vdash A \times X[k] : * \end{array}$$

This yields the constructors $x : \mathbf{1} \vdash \alpha_1 : \mathrm{Vec}\, A[0]$ and $k : \mathbb{N}, x : A \times \mathrm{Vec}\, A[k] \vdash \alpha_2 : \mathrm{Vec}\, A[k+1]$, which are usually called nil and cons.

In the talk, we show that the above calculus has indeed the structure we are aiming for, in the sense that types in context give rise to a split fibration, if we employ the expected equations on substitutions ($A[\mathrm{id}] = A$, $x_i[f] = f_i$, etc.). This allows us to conveniently define actions of types on terms, and a reduction relation. For now, we have no proof of type preservation etc.

References

[1] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In *Proc. of POPL '13*, pp. 27–38. ACM, 2013.

[2] T. Hagino. A typed lambda calculus with categorical type constructors. In *Proc. of CTCS '87*, v. 283 of *Lect. Notes in Comput. Sci.*, pp. 140–157. Springer, 1987.

[3] M. Hamana and M. Fiore. A foundation for GADTs and inductive families: Dependent polynomial functor approach. In *Proc. of WGP '11*, pp. 59–70. ACM, 2011.

The Decision Problem for Linear Tree Constraints

Sabine Bauer and Martin Hofmann, LMU Munich

Institut für Informatik, Ludwig-Maximilians-Universität München, Germany
{hofmann,sabine.bauer}@ifi.lmu.de

We present new results on a constraint satisfaction problem arising from the inference of resource types.

Linear constraints were introduced by Hofmann and Jost in the context of type-based amortized resource analysis by the potential method [5] where it was applied to functional programs. The constraint systems appearing in this system and subsequent ones has finitely many variables and can be reduced to linear programming. Later, Hofmann and Rodriguez extended type-based amortized resource analysis to object-oriented programs [8],[6] which led to constraints involving infinite lists or trees whose entries are numerical variables. Therefore, a straightforward reduction to linear programming is no longer an option. However, the constraint systems exhibit enough regularity that a heuristic procedure developed by Hofmann and Rodriguez allowed to find solutions in many cases.

In the case of infinite lists the constraint systems can be simply described as follows. One has finitely many unknowns ranging over infinite lists (sequences) of nonnegative rational numbers including ∞. Terms and constraints are built from those by addition ($+$) and comparison ($\leq$) understood pointwise, and the tail function (tl) that removes the first element of a sequence. Furthermore, constraint systems may contain ordinary linear arithmetic constraints over the nonnegative rationals including infinity where the head function (hd) maps sequences to numbers. For example, the following constraint system is solvable and has the (not unique) solution $\mathbf{y}$ a constant list and $\mathbf{x}$ an exponentially decreasing list and $\mathbf{z}$ a Fibonacci list with an additional linear summand. We have the (arithmetic and list) constraints

$$\begin{aligned} hd(\mathbf{x}) &= 2, & tl(\mathbf{y}) &\geq \mathbf{y}, \\ hd(tl(\mathbf{x})) &= 5, & \mathbf{z}+\mathbf{z} &\leq tl(\mathbf{z}), \\ hd(\mathbf{y}) &\geq 1, & tl(tl(\mathbf{x})) &\geq tl(\mathbf{x})+\mathbf{x}+tl(\mathbf{y})+tl(\mathbf{y}). \\ hd(\mathbf{z}) &\geq 2, \end{aligned}$$

and the solutions

$$\begin{aligned} &\mathbf{y} = 1, 1, ..., \\ &\mathbf{x} = 2, 5, 9, 16, 27, 55, .., n-1, n, n(n-1)+2, ..., \\ &\mathbf{z} = 2, 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, ..., \frac{1}{2^n}, ... \end{aligned}$$

In this paper we report progress on the general question of solvability and decidability of these constraint systems. First, in the case of infinite lists of numerical variables we can in many cases draw a connection to rational generating functions and in particular see that certain resource behaviours cannot arise as solutions to type inference problems. A concrete example is the harmonic series because it has a logarithmic generating function [4].

We also show that the general constraint satisfaction problem as formulated by Hofmann and Rodriguez admits a reduction from the Skolem-Mahler-Lech problem [7, 2] whose decidability status is as yet unknown but at least NP-hard. Recall that the Skolem-Mahler-Lech problem

asks whether a sequence defined by a linear recurrence with integer coefficients (like Fibonacci) contains zero.

However, we were able to better delineate the image of the translation from type inference to the constraint satisfaction problem and thus identified a subproblem which—if solved—can still be used for type-based resource inference but has better algorithmic properties.

More precisely, we can show that the characteristic polynomials of the linear recurrences corresponding to the identified fragment always have a dominating zero (eigenvalue) which allows for a decision procedure based on comparison of growth rates. There are still several fine points to overcome which result from the fact that we are dealing with inequations rather than equations.

We then also give a heuristic reduction from the case of general trees to infinite lists which carries type-based amortized resource analysis considerably further than it was previously the case. In particular, we are able to analyse programs with superpolynomial resource consumption and also have a more intrinsic characterisation of the limitations of the analysis.

In currently ongoing work, we try to go beyond the question of mere satisfiability of constraint systems and to extract minimal solutions and also to treat trees in general not merely by translating them to lists.

References

[1] J. Bell and S. Gerhold. The positivity set of a recurrence sequence. arXiv:0506.398, 2005.

[2] V. Blondel and N. Portier. The presence of a zero in an integer linear recurrent sequence is NP-hard to decide. *Lin. Alg. and Its Appl.*, 351–351:91–98, 2002.

[3] G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*. Oxford Univ. Press, 2003.

[4] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1994.

[5] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Conf. Record of 30th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '03*, pp. 185–197. ACM, 2003.

[6] M. Hofmann and D. Rodriguez. Linear constraints over infinite trees. In *Proc. of 18th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2012*, v. 7180 of *Lect. Notes in Comput. Sci.*, pp. 343–358. Springer, 2012.

[7] J. Ouaknine and J. Worrell. Positivity problems for low-order linear recurrence sequences. arXiv:1307.2779, 2013.

[8] D. Rodriguez. *Amortized Analysis for object oriented programs*. PhD thesis, LMU München, 2012.

A Presheaf Model of Parametric Type Theory

Jean-Philippe Bernardy, Thierry Coquand and Guilhem Moulin

Dept. of Comput. Sci and Engin., Chalmers University of Technology and University of Gothenburg, Sweden
{bernardy,coquand,mouling}@chalmers.se

Abstract

We propose a new type theory with internalized parametricity. Compared to previous similar proposals, this version comes with a denotational semantics which is a refinement of the standard presheaf semantics of dependent type theory. Further, this presheaf semantics is a refinement of the one used to interpret nominal sets with restriction. The present calculus is a candidate for the core of a proof assistant with internalized parametricity.

Reynolds's abstraction theorem can be stated in a purely syntactical way: for instance, if a function f has type $(A : \star) \to A \to A$ — the type of the polymorphic identity — then the proposition $(A : \star) \to (P : A \to \star) \to (x : A) \to P\,x \to P\,(f\,A\,x)$ holds. However this result is not provable internally, *i.e.*, $(f : (A : \star) \to A \to A) \to (A : \star) \to (P : A \to \star) \to (x : A) \to P\,x \to P\,(f\,A\,x)$ is not provable. Several attempts have been made for designing an extension of dependent type theory in which such an internal form of parametricity holds. We propose another such system here. Our technical contributions are as follows:

- We present a type theory which internalizes parametricity and can be seen as a simplification and generalization of the systems of [1, 2]
- We provide a *denotational* semantics, in the form of a presheaf model, for this type theory. This model is a refinement of the presheaf semantics used to interpret nominal sets with restrictions [3, 4].

Syntax

We assume a special symbol '0', and a countable infinite set $\mathbb{I}$ of other symbols, called *colors*. The metasyntactic variables $i, j, \ldots$ range over colors, while φ range over $\mathbb{I} \cup \{0\}$. The main innovation of the type theory presented here is that terms may depend on (a finite number of) colors. We add the following constructions to the usual syntax of lambda calculi:

$$a, p, t, A, P, T \coloneqq \ldots \mid (a,_i\, p) \mid (x : A) \times_i P \mid A \ni_i a \mid a{\cdot}i$$

Remark. Here is some intuition for these new constructions:

- Any type is associated with a predicate for every color. The type $A \ni_i a$ expresses that a satisfies the parametricity predicate associated with the type A on color i. For each term a and color i, the term $a\,(i\,0)$ is the erasure of i in a. It is defined by induction on a and can be understood as a realizer of a.
- The term $a{\cdot}i$ yields a proof of $A \ni_i a\,(i\,0)$.
- The forms $(a,_i\, p)$ and $(x : A) \times_i P$ allow to locally associate parametricity proofs with a given realizer.

We index typing judgements with the set of free colors; our new constructions are typed and converted as follows:

$$\textsc{In-Abs}\ \frac{\Gamma \vdash a : A\,(i\,0) \qquad \Gamma \vdash p : A \ni_i a}{\Gamma, i : \mathbb{I} \vdash (a,_i p) : A} \qquad \frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash P}{\Gamma, i : \mathbb{I} \vdash (x : A) \times_i P}\ \textsc{In-Pred}$$

$$\textsc{Out}\ \frac{\Gamma, i : \mathbb{I} \vdash A \qquad \Gamma \vdash a : A\,(i\,0)}{\Gamma \vdash A \ni_i a} \qquad \frac{\Gamma, i : \mathbb{I} \vdash a : A}{\Gamma \vdash a{\cdot}i : A \ni_i a\,(i\,0)}\ \textsc{Color-Elim}$$

$$(a,_i p){\cdot}i = p \qquad ((x : A) \times_i P[x]) \ni_i a = P[a]$$

$$t = (t\,(i\,0),_i t{\cdot}i) \qquad T = (x : T\,(i\,0)) \times_i (T \ni_i x)$$

Unlike previous type theories with internalized parametricity, the types $\star \ni_i A$ and $A \to \star$ are not convertible but *isomorphic*. The same goes for $((x : A) \to B[x]) \ni_i f$ and $(x : A) \to (x' : A \ni_i x) \to B[(x,_i x')] \ni_i (f\,x)$. However, in our system one can use parametricity generically via $\lambda A.\,\lambda a.\,a \cdot i \ : \ (A : \star) \to (x : A) \to A \ni_i x$. In particular, the proposition given in the introduction is provable by $\lambda f.\,\lambda A.\,\lambda P.\,\lambda a.\,\lambda p.\,(f\,(A \times_i P)\,(a,_i p)){\cdot}i$.

Presheaf model

We say that a function $f : I \to J \cup \{0\}$ is a *color map*, and note $f : I \to J$, if $i_1 = i_2$ for any $i_1, i_2 \in I$ with $f(i_1) = f(i_2) \in J$. We consider the category $\mathbf{pI}$ of finite color sets and color maps. We use a refined presheaf on $\mathbf{pI}^{\mathrm{op}}$ by requiring two further conditions (without this refinement, it is not clear how to validate the equality $((x : A) \times_i P[x]) \ni_i a = P[a]$):

1. for any object I, $F(I)$ is a set of I-elements, *i.e.*, of tuples indexed by the subsets of I;
2. for any projection map $\alpha : I \to I_\alpha$, the restriction map $F(I) \to F(I_\alpha)$, $u \mapsto u\alpha$ is the projection operation, *i.e.*, $u\alpha_J = u_J$ for any $J \subseteq I$.

A context $\Gamma \vdash$ is interpreted by a (usual) presheaf on $\mathbf{pI}^{\mathrm{op}}$.

A type $\Gamma \vdash A$ is interpreted by an I-set $A\rho$ for each object I and $\rho \in \Gamma(I)$, together with restriction maps $A\rho \to A(\rho f)$, $u \mapsto uf$ if $f : I \to J$ satisfying $u1 = u$ and $(uf)g = u(fg)$ for any $g : J \to K$. Furthermore the map $A\rho \to A(\rho\alpha)$, $u \mapsto u\alpha$ is the projection operation.

A term $\Gamma \vdash a : A$ is interpreted by a I-element $a\rho \in A\rho$ for each object I and $\rho \in \Gamma(I)$, such that $a\rho f = a(\rho f)$ for any $f : I \to J$.

If $\Gamma \vdash$ and $\Gamma \vdash A$ we define the interpretation of $\Delta = \Gamma, x : A$ by taking $\langle \rho, x = u \rangle \in \Delta(I)$ to mean $\rho \in \Gamma(I)$ and $u \in A\rho$. The restriction map is defined by $\langle \rho, x = u \rangle f = \langle \rho f, x = uf \rangle$.

If $\Gamma \vdash$ we define the interpretation of $\Delta = \Gamma, i : \mathbb{I}$ by taking $[\rho, i = \varphi] \in \Delta(I)$ to mean either $\varphi = 0$ and $\rho \in \Gamma(I)$, or $\varphi = j \in I$ and $\rho \in \Gamma(I \backslash \{j\})$.

References

[1] J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *Proc. of 27th Ann. IEEE Symp. on Logic in Comput. Sci., LICS '12*, pp. 135–144. IEEE CS, 2012.

[2] J.-P. Bernardy and G. Moulin. Type-theory in color. In *Proc. of 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '13*, pp. 61–72. ICFP, 2013.

[3] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In *Proc. of 19th Int. Conf. on Types for Proofs and programs, TYPES '13*, v. 26 of *Leibniz Int. Proc. in Inform.*, pp. 107–128, 2014.

[4] A. M. Pitts. An equivalent presentation of the Bezem-Coquand-Huber category of cubical sets. arXiv:1401.7807, 2014.

Guarded Dependent Type Theory with Coinductive Types

Aleš Bizjak[1], Hans Bugge Grathwohl[1], Ranald Clouston[1],
Rasmus E. Møgelberg[2] and Lars Birkedal[1]

[1] Department of Computer Science, Aarhus University, Denmark
[2] IT University of Copenhagen, Denmark
{abizjak|hbugge|ranald.clouston|birkedal}@cs.au.dk, mogel@itu.dk

Modern implementations of intensional dependent type theories, such as Coq, Agda, and Idris, have been used successfully for programming and proving in many projects. However they offer limited support for *coinductive* types.

For programming, the challenge is to ensure that functions on coinductive types are well-defined; that is, productive with unique solutions. Syntactic guardedness checks, as used for example in Coq, ensure productivity by requiring that recursive calls be nested directly under a constructor, but such checks exclude many valid definitions, particularly in the presence of higher-order functions.

For proving, the challenge is to reconcile the extensional nature of coinductive types, which are characterised by observations, with intensional equality. For example, the natural notion of bisimulation on streams does not coincide with inhabitation of the identity type. Thus one is forced to introduce a non-standard notion of equality for coinductive types, which will typically not be a congruence [4]. It would be preferable if one could instead directly use the identity relation of the type theory.

For the programming challenge, a *type-based* approach to guarded recursion, more flexible than syntactic checks, was suggested by Nakano [6]. A new modality $\triangleright$, called 'later', allows us to distinguish between data we have access to now, and data which we have only later. $\triangleright$ must guard self-reference in type definitions, so for example *guarded streams* of natural numbers are described by the guarded recursive equation $\mathsf{Str}^g_{\mathbb{N}} \simeq \mathbb{N} \times \triangleright \mathsf{Str}^g_{\mathbb{N}}$ asserting that stream heads are available now, but tails only later.

Using $\triangleright$ alone, however, enforces a discipline more rigid than productivity. For example, all stream functions must be *causal* [3], so elements of the result must not depend on deeper elements of the argument, ruling out the 'every other' function that returns every second element of a given stream. This limitation motivated the introduction of *clock quantifiers* by Atkey and McBride [1], which permit $\triangleright$ to be eliminated in a controlled way. In earlier work [2] we recast these quantifiers as the unary type-former $\Box$, called 'constant'. Moreover such type-formers give rise to types whose denotation is exactly that of standard coinductive types [5, Theorem 2]. Hence $\Box$ allows us to program and prove with 'real' coinductive types, with guarded recursion providing the 'rule format' for their manipulation. For example, the coinductive type of streams of natural numbers can be defined simply as $\mathsf{Str}_{\mathbb{N}} \triangleq \Box \mathsf{Str}^g_{\mathbb{N}}$.

In previous work [2], we introduced the *guarded λ-calculus*, $\mathtt{g}\lambda$, a simply typed lambda calculus with guarded and coinductive types, and a logic, $L\mathtt{g}\lambda$, as a separate layer, to prove properties of elements. The logic allowed us to prove properties of coinductive types, but not in the integrated fashion supported by dependent type theories. In this talk we introduce guarded dependent type theory, gDTT, which supports programming and proving with guarded and coinductive dependent types.

For the proving challenge, our approach is to use guarded recursion to prove intensional equality of terms of guarded recursive types. From this we can derive equalities of terms of

coinductive types, thus circumventing the need for a non-standard equality type for coinductive types.

One of the key challenges in designing gDTT is coping with elements that are only available later, i.e., elements of types of form $\triangleright A$. With gλ we had the term formation rules

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\, t : \triangleright A} \qquad \frac{\Gamma \vdash f : \triangleright(A \to B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash f \circledast t : \triangleright B}$$

The first rule allows us to make later use of data that we have now. The second intuitively says that if f will later be a function mapping A to B, and we will later have an element t of A, then, later, we can apply f to t to get an element of B. Let us consider the $\circledast$ rule for dependent types, where function spaces are generalised to Π-types. Supposing that f has type $\triangleright(\Pi x : A.B)$, what type should $f \circledast t$ have? If B depends on x, it does not make sense to follow the simply-typed approach and use $\triangleright B$. But since t has type $\triangleright A$, and not A, we cannot substitute t for x. Intuitively, t will eventually reduce to some value $\mathsf{next}\, u$, and so the resulting type should be $\triangleright B[u/x]$. But if t is an open term we may not be able to perform this reduction, so cannot type our term. To address this issue we introduce a new notion, of *delayed substitution*, allowing us to express the resulting type as $\triangleright[x \leftarrow t].B$. Definitional equality rules allow us to simplify this type when t is a value, i.e., $\triangleright[x \leftarrow \mathsf{next}\, u].B \simeq \triangleright B[u/x]$ as expected.

The novel features of gDTT allow us to address both the programming and proving challenges posed by coinductive types. To illustrate, consider addition of two guarded streams:

$$\mathsf{plus}^g \triangleq \begin{array}{l}\mathsf{fix}\,\phi.\lambda(xs : \mathsf{Str}^g_{\mathbb{N}})(ys : \mathsf{Str}^g_{\mathbb{N}}).\\ \quad \mathsf{cons}^g\,((\mathsf{hd}^g\,xs) + (\mathsf{hd}^g\,ys))\;(\phi \circledast \mathsf{tl}^g\,xs \circledast \mathsf{tl}^g\,ys)\end{array} \;:\; \mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}}$$

Here hd^g takes the head of a stream, and tl^g its tail (recalling that tails of guarded streams have type $\triangleright\,\mathsf{Str}^g_{\mathbb{N}}$). This function as whole is defined by guarded recursion, so ϕ has type $\triangleright(\mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}})$. The $\circledast$-application $\phi \circledast \mathsf{tl}^g\,xs \circledast \mathsf{tl}^g\,ys$ has type $\triangleright\,\mathsf{Str}^g_{\mathbb{N}}$, so the $\mathsf{cons}^g(\cdots)$ sub-expression has type $\mathsf{Str}^g_{\mathbb{N}}$.

With gDTT we can prove the commutativity of addition on streams. Assuming that c is a proof term witnessing commutativity of $+$ on natural numbers, and that $\mathsf{p}\eta$ is a proof term yielding equality of two pairs when given equality proofs of their respective components, we can construct a proof:

$$p \triangleq \begin{array}{l}\mathsf{fix}\,\phi.\lambda\,(xs, ys : \mathsf{Str}^g_{\mathbb{N}})\,.\\ \quad \mathsf{p}\eta\;(c\,(\mathsf{hd}^g\,xs)\,(\mathsf{hd}^g\,ys))\;(\phi \circledast \mathsf{tl}^g\,xs \circledast \mathsf{tl}^g\,ys)\end{array} \;:\; \Pi(xs, ys : \mathsf{Str}^g_{\mathbb{N}}).\mathsf{Id}_{\mathsf{Str}^g_{\mathbb{N}}}(\mathsf{plus}^g\,xs\,ys, \mathsf{plus}^g\,ys\,xs).$$

where Id is the identity type-former. Note that p is again defined by guarded recursion, and that it is quite simple: it says that to show commutativity of plus^g we proceed by using commutativity of $+$ on the heads, then continuing recursively on the tails. While p itself is simple, showing that p indeed has the right type makes use of the new features in gDTT. For example, the subterm $\phi \circledast \mathsf{tl}^g\,xs \circledast \mathsf{tl}^g\,ys$ cannot be typed without use of the generalised $\triangleright$ carrying a delayed substitution.

We can lift the function plus^g to a function on *coinductive* streams $\mathsf{plus} : \mathsf{Str}_{\mathbb{N}} \to \mathsf{Str}_{\mathbb{N}} \to \mathsf{Str}_{\mathbb{N}}$. Likewise, using the new features of gDTT, we can turn the proof p of commutativity of plus^g into one for plus, i.e., construct a term of type $\Pi(xs, ys : \mathsf{Str}_{\mathbb{N}}).\mathsf{Id}_{\mathsf{Str}_{\mathbb{N}}}(\mathsf{plus}\,xs\,ys, \mathsf{plus}\,ys\,xs)$.

Thus we can use guarded recursion to program and prove properties on guarded streams, and then convert these into programs and proofs for coinductive streams.

This abstract is based on a recently submitted paper.

References

[1] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *Proc. of ICFP '13*, pp. 197–208. ACM, 2013.

[2] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal. Programming and reasoning with guarded recursion for coinductive types. In *Proc. of FoSSaCS 2015*, v. 9034 of *Lect. Notes in Comput. Sci.*, pp. 279–294.Springer, 2015.

[3] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Proc. of LICS '11*, pp. 257–266. IEEE CS, 2011.

[4] C. McBride. Let's see how things unfold: Reconciling the infinite with the intensional. In *Proc. of CALCO '09*, v. 5728 of *Lect. Notes in Comput. Sci.*, pp. 113–126. Springer, 2009.

[5] R. E. Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Proc. of CSL-LICS '14*, art. 71. ACM, 2014.

[6] H. Nakano. A modality for recursion. In *Proc. of LICS '00*, pp. 255–266. IEEE CS, 2000.

Rewrite Semantics for Guarded Recursion with Universal Quantification over Clocks

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

[1] Department of Computer Science, Aarhus University, Denmark, abizjak@cs.au.dk
[2] IT University of Copenhagen, Denmark, mogel@itu.dk

Guarded recursion [7] is an approach to solving recursive type equations where the type variable appears *guarded* by a $\blacktriangleright$ (pronounced "later") modal type operator. In particular the type variable could appear positively or negatively or both, e.g. the equation $D = 1 + \blacktriangleright(D \to D)$ has a unique solution [4]. On the term level the *guarded fixed point combinator* $\mathsf{fix}_A : (\blacktriangleright A \to A) \to A$ satisfies the equation $f\,(\mathsf{next}\,(\mathsf{fix}_A f)) = \mathsf{fix}_A f$ for any $f : \blacktriangleright A \to A$. Here $\mathsf{next} : A \to \blacktriangleright A$ is an operation that "freezes" an element that we have available now so that it is only available in the next time step.

One situation where guarded recursive types are useful is when faced with an unsolvable type equation. These arise for example when modelling advanced programming languages. In this case a solution to a guarded version of the equation often turns out to suffice, cf. [4].

But guarded recursive versions of polymorphic type equations are also useful in type theory, even in settings where inductive and coinductive solutions to these equations are assumed to exist. To see this, consider the coinductive type of streams Str, i.e., the final coalgebra for the functor $\mathbb{S}(X) = \mathbb{N} \times X$. The proof assistants Coq and Agda allow programmers to construct streams using recursive definitions, but to ensure normalisation (and thereby consistency), these recursive definitions must be *productive*, i.e., one must be able to compute the n first elements of a stream in finite time. Coq and Agda inspect recursive definitions for productivity by a syntactic property that does not interact well with higher-order functions.

Using the type of *guarded streams* Str_g, i.e., the unique type satisfying the equation $\mathsf{Str}_g = \mathbb{N} \times \blacktriangleright \mathsf{Str}_g$, one can encode productivity in types: a productive recursive stream definition is exactly a term of type $\blacktriangleright \mathsf{Str}_g \to \mathsf{Str}_g$. To combine the benefits of coinductive and guarded recursive types, Atkey and McBride [3] suggested a simply typed calculus with clock variables κ representing time streams, each with associated $\blacktriangleright^\kappa$ type constructors, and universal quantification over clocks $\forall\kappa$. If we think of the type τ as being time-indexed along κ, then the type $\forall\kappa.\tau$ contains only elements which are available for all time steps. The relationship between the two notions of streams can then be captured by the encoding of the coinductive stream type as $\mathsf{Str} = \forall\kappa.\mathsf{Str}_g^\kappa$. This encoding works for a general class of coinductive types including those given by polynomial functors, and these results were since extended to the dependently typed setting by Møgelberg [6]. In both cases the encodings were proved sound with respect to a denotational model and no rewrite semantics was given. This work is strongly related to sized types [2, 1].

Rewrite semantics In this talk we will present our ongoing work on giving computational meaning to guarded recursion and $\forall\kappa$. One of the challenges is to give operational meaning to a series of type isomorphisms assumed in both Atkey and McBride [3] and Møgelberg [6]. These type isomorphisms are crucial to prove correctness of the encodings of coinductive types.

One of these isomorphisms is $\forall\kappa.\, \blacktriangleright^\kappa \tau \cong \forall\kappa.\tau$ stating that time steps on universally quantified clocks can be ignored. We suggest a partial eliminator for $\blacktriangleright^\kappa$ which suffices to encode the isomorphism. Note that an unrestricted eliminator would cause an inconsistency: if we allow a term of type $\blacktriangleright^\kappa \tau \to \tau$ for all τ, the fixed point of these maps give inhabitants of any type.

Another type isomorphism is $\forall\kappa.\tau \cong \tau$, valid whenever κ is not free in τ. This is used in the encoding of coinductive types to give the head map type $\mathsf{Str} \to \mathbb{N}$ rather than $\mathsf{Str} \to \forall\kappa.\mathbb{N}$. There is a natural term $\lambda x.\Lambda\kappa.x$ of type $\tau \to \forall\kappa.\tau$ and we want to reflect in the calculus that this term has an inverse.[1] We suggest a new term construct af (**a**pply at **f**resh) with the typing rule

$$\frac{\Delta \mid \Gamma \vdash t : \forall\kappa.\tau \qquad \kappa \notin \tau}{\Delta \mid \Gamma \vdash \mathsf{af}\, t : \tau}$$

Here Δ is a clock context (a finite set of clocks) and Γ is an ordinary context of term variables whose types only contain clocks in the set Δ. Intuitively, $\mathsf{af}\, t$ should generate a fresh name for a clock variable and apply t to it. Since fresh name generation is an effect it is non-trivial to give semantics to it in type theory, see e.g. [9, 5]. In this setting, however, we believe that it suffices to have a rule $\mathsf{af}(\Lambda\kappa.t) \mapsto t$ applicable only when κ is not free in t, along with a family of "commuting conversions" analogous to those of the ν construct of Nominal System T [8].

We will show that just the two type isomorphisms listed above suffice for constructing the remaining type isomorphisms listed in [3, 6]. In particular we can show $\forall\kappa$ commutes over binary sums, if those sums are encoded using dependent sums, universes and booleans in the standard way.

Clock synchronisation In the talk we will also address the problem of clock synchronisation, which was disallowed in previous work. This restriction arose in the elimination rule for $\forall\kappa$:

$$\frac{\Delta \mid \Gamma \vdash t : \forall\kappa.\tau \qquad \kappa' \in \Delta}{\Delta \mid \Gamma \vdash t[\kappa'] : \tau[\kappa'/\kappa]} \tag{1}$$

which in [3, 6] had freshness side conditions on κ'. The reason for such a restriction was that the models did not support clock *substitution*, only clock *permutation*. In this talk we present a new model that does allow clock substitution, and thus proves the unrestricted version of (1) sound. This greatly simplifies the language and the rewrite semantics.

References

[1] A. Abel and B. Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *Proc. of ICFP '13*, pp. 185–196. ACM, 2013.

[2] A. Abel and A. Vezzosi. A formalized proof of strong normalization for guarded recursive types. In *Proc. of APLAS 2014*, v. 8858 of *Lect. Notes in Comput. Sci.*, pp. 140–158. Springer, 2014.

[3] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *Proc. of ICFP '13*, pp. 197–208. ACM, 2013.

[4] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods in Comput. Sci.*, 8(4:1), 2012.

[5] J. Cheney. A dependent nominal type theory. *Log. Methods in Comput. Sci.*, 8(1:8), 2012.

[6] R. E. Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Proc. of CSL-LICS 2014*, art. 71. ACM, 2014.

[7] H. Nakano. A modality for recursion. In *Proc. of LICS '00*, pp. 255–266. IEEE CS, 2000.

[8] A. M. Pitts. Structural recursion with locally scoped names. *J. of Funct. Program.*, 21(3):235–286, 2011.

[9] A. M. Pitts, J. Matthiesen, and J. Derikx. A dependent type theory with abstractable names. In *Proc. of LSFA 2014*, v. 312 of *Electron. Notes in Theor. Comput. Sci.*, pp. 19–50. Elsevier, 2015.

[1] Compare this to the situation in, e.g. System F where the types $\mathbb{N}$ and $\forall\alpha.\mathbb{N}$ should be isomorphic, but the $\beta\eta$-laws are not sufficient to derive this; parametricity is needed.

Typed Realizability for First-Order Classical Analysis

Valentin Blot

Dept. of Computer Science, University of Bath, United Kingdom
v.blot@bath.ac.uk

In realizability we associate to each formula a set of programs which behave like a proof of the formula and that we call its realizers. This allows to give computational content to the axioms of a theory, and to choose quite freely the programming language, independently from the logical system. It is even possible to consider untyped programming languages, as was the case in the first realizability model from Kleene [7], in which a realizer may be any recursive function. It is however still possible to consider a typed language, as did Kreisel in his modified realizability model [8]. Both models from Kleene and Kreisel gave computational interpretation to Heyting arithmetic, the intuitionistic variant of Peano arithmetic.

Gödel's negative translation [4] allowed for the so-called indirect interpretations of classical logic through intuitionistic interpretations of negatively translated classical proofs. This allowed to interpret full Peano arithmetic using the realizability models of Kleene and Kreisel. Much later, Griffin's discovery [6] that the `call/cc` control operator could be typed with the law of Peirce opened the possibility for a direct interpretation of classical logic, using programming languages with control features. Following this path, Parigot defined the $\lambda\mu$-calculus [12], a language for Gentzen's classical sequent calculus for which Selinger axiomatized the universal categorical model [13]. On another side, Krivine considered untyped λ-calculus extended with the `call/cc` operator to give a realizability interpretation to classical second-order Zermelo-Frænkel set theory [9], later extended to handle the axiom of dependent choice [10, 11].

In this talk I will present a direct realizability interpretation for first-order classical logic which, contrary to Krivine's and similarly to Kreisel's (but for classical logic), uses typed programs as realizers. The classical proofs are interpreted in a categorical model of the language μPCF, which is a combination of the functional Turing-complete language PCF with the control features of call-by-name $\lambda\mu$-calculus. A careful analysis of relativization motivates the distinction between negative and positive formulas, the former having both a truth value and a falsity value which are orthogonal to each other as in Krivine's work, but the latter having only a truth value, which is not closed by double-orthogonality. A suitable and non-restrictive requirement on the classical proofs (right structural rules are forbidden for positive formulas) allows to prove the soundness of the interpretation. The model is validated by proving that the usual terms of Gödel's system T realize the axioms of Peano arithmetic. Concerning extraction of programs from proofs, Friedman's trick is implemented through the use of an external μ-variable rather than through the replacement of the $\bot$ formula by an existential statement, which allows for a simpler and more effective interpretation in some models (typically the models based upon game semantics). The choice of having a free μ-variable κ in the realizers may be seen as a way to avoid Friedman's A-translation. Indeed, in Friedman's original work, the translation is obtained by replacing each basic predicate P with the disjunction $P \vee A$. In classical sequent calculus, the right-hand context is meant to be interpreted as a disjunction, so adding a fixed μ-variable to this context corresponds to applying Friedman's translation on programs instead of proofs. This variable is also used to define the orthogonality relation between our truth and falsity values.

Interpreting the axiom of dependent choice in a classical setting is much more complicated than interpreting arithmetic. To interpret it, Spector defined the bar recursor [14] and used it in Gödel's Dialectica interpretation [5] (a computational interpretation similar to realizability). A more uniform version was then used to give an indirect realizability interpretation of countable choice [1], and a version with an implicit termination condition was later defined and used to interpret, still in an indirect realizability setting, the double-negation shift principle and therefore the axiom of dependent choice [2].

In the direct interpretation described in the talk and under some assumptions on the model of μPCF, the bar recursion operator [2] realizes the axiom of dependent choice, similarly to what was done in a previous work [3]. The particular implementation of Friedman's trick then allows to obtain an extraction result on Π^0_2 formulas provable in classical analysis (Peano arithmetic + the axiom of dependent choice).

This work has been submitted recently as a journal paper.

References

[1] S. Berardi, M. Bezem, and T. Coquand. On the computational content of the axiom of choice. *J. of Symb. Log.*, 63(2):600–622, 1998.

[2] U. Berger and P. Oliva. Modified bar recursion and classical dependent choice. In *Proc. of 2001 Ann. Europ. Summer Meeting of ASL, Logic Colloquium '01*, v. 20 of *Lect. Notes in Logic*, pp. 89–107. A. K. Peters, Ltd., 2005.

[3] V. Blot and C. Riba. On bar recursion and choice in a classical setting. In *Proc. of 11th Asian Symp. on Programming Languages and Systems*, v. 8301 of *Lect. Notes in Comput. Sci.*, pp. 349–364. Springer, 2013.

[4] K. Gödel. Zur intuitionistischen Arithmetik und Zahlentheorie. *Ergebnisse eines mathematischen Kolloquiums*, 4:34–38, 1933.

[5] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12(3-4):280–287, 1958.

[6] T. Griffin. A formulae-as-types notion of control. In *Conf. Record of 17th Ann. Symp. on Principles of Programming Languages, POPL '90*, pages 47–58. ACM Press, 1990.

[7] S. C. Kleene. On the interpretation of intuitionistic number theory. *J. of Symb. Log.*, 10(4):109–124, 1945.

[8] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In *Proc. of Coll. on Constructivity in Mathematics (Amsterdam, 1957)*, *Studies in Logic and the Foundations of Mathematics*, pp. 101–128. North Holland, 1959.

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

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

[11] J.-L. Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009.

[12] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. of 3rd Int. Conf. on Logic Programming and Automated Reasoning, LPAR '92*, v. 624 of *Lect. Notes in Comput. Sci.*, pp. 190–201. Springer, 1992.

[13] P. Selinger. Control categories and duality: on the categorical semantics of the $\lambda\mu$ calculus. *Math. Struct. in Comput. Sci.*, 11(2):207–260, 2001.

[14] C. Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In *Recursive Function Theory*, v. 5 of *Proc. of Symp. in Pure Math.*, pp. 1–27. AMS, 1962.

Quotienting the Delay Monad by Weak Bisimilarity

James Chapman, Tarmo Uustalu and Niccolò Veltri

Institute of Cybernetics at Tallinn University of Technology, Estonia
{james|tarmo|niccolo}@cs.ioc.ee

The delay datatype was introduced by Capretta [2] as a means to incorporate general recursion to Martin-Löf type theory and it is useful in this setting for modeling non-terminating behaviours. This datatype is a (strong) monad and constitutes a constructive alternative to the maybe monad. For a given set X, each element of $\mathsf{D}\,X$ is a possibly infinite computation that returns a value of X, if it terminates. We define $\mathsf{D}\,X$ as a coinductive type by the rules

$$\frac{}{\mathsf{now}\,x : \mathsf{D}\,X} \qquad \frac{c : \mathsf{D}\,X}{\mathsf{later}\,c : \mathsf{D}\,X}$$

Weak bisimilarity is defined in terms of convergence. This binary relation between $\mathsf{D}\,X$ and X relates a terminating computation to its value and is inductively defined by the rules

$$\frac{}{\mathsf{now}\,x \downarrow x} \qquad \frac{c \downarrow x}{\mathsf{later}\,c \downarrow x}$$

Two computations are weakly R-bisimilar if they differ by a finite number of application of the constructor later, i.e., they either converge to R-related values or diverge. Weak R-bisimilarity is defined coinductively by the rules

$$\frac{c \downarrow x \quad xRx' \quad c' \downarrow x'}{c \approx_R c'} \qquad \frac{c \approx_R c'}{\mathsf{later}\,c \approx_R \mathsf{later}\,c'}$$

Just as D, one would expect $\hat{\mathsf{D}}$ defined by $\hat{\mathsf{D}}\,X = \mathsf{D}\,X/{\approx_X}$ (by writing $\approx_X$ we mean $\approx_{\equiv_X}$) to also be a (strong) monad. Morally, this ought to be the case, but there are issues with defining the multiplication, having to do with quotients in type theory.

One possible approach to quotients is to avoid them altogether as first-class entities, by working with setoids. This is unproblematic, we can define $\hat{\mathsf{D}}$ as a functor on **Setoid** by $\hat{\mathsf{D}}\,(X, R) = (\mathsf{D}\,X, \approx_R)$ and it is then also a (strong) monad. This route was taken by Capretta [2] and the same was done also by Benton et al. [1].

If we want to use quotients as sets, then we need to add them to type theory. We follow the approach of Hofmann [3]. We postulate, for any set X and equivalence relation R on X the existence of the following data: a set X/R, a constructor $\mathsf{abs} : X \to X/R$ with a proof $\mathsf{sound} : \Pi x, x' : X.\, \Pi p : xRx'.\ \mathsf{abs}\,x \equiv \mathsf{abs}\ x'$, for any predicate P on X/R, a dependent eliminator $\mathsf{lift} : \Pi f : \Pi x : X.\, P\,(\mathsf{abs}\,x).\, \mathsf{compat}\,f \to \Pi q : X/R.\, P\,q$ where $\mathsf{compat}\,f = \Pi x, x' : X.\ \Pi p : xRx'.\, \mathsf{subst}\,P\,(\mathsf{sound}\,p)(f\,x) \equiv f\,x'$ and a proof of the beta-rule $\mathsf{lift}_\beta : \mathsf{lift}\,f\,p\,(\mathsf{abs}\,x) \equiv f\,x$. A special case of quotients are so-called squash types, which are quotients by the total relation. We write $\|X\|$ for $X/\top$. We call a set X a proposition, if $\Pi x, x' : X.\, x \equiv x'$. Squash types are propositions.

For $\hat{\mathsf{D}}$ as a functor on **Set** to be a monad, we have to have a multiplication $\mu_X : \hat{\mathsf{D}}\,(\hat{\mathsf{D}}\,X) \to \hat{\mathsf{D}}\,X$. For this, a crucial element needed is a function $\psi : \mathsf{D}\,(\mathsf{D}\,X/{\approx_X}) \to \mathsf{D}\,(\mathsf{D}\,X)/\mathsf{D}{\approx_X}$. This seems difficult, but there is a canonical function $\theta = \mathsf{lift}\,(\mathsf{D}\,\mathsf{abs})\,(\ldots) : \mathsf{D}\,(\mathsf{D}\,X)/\mathsf{D}{\approx_X} \to$

$\mathsf{D}\,(\mathsf{D}\,X/{\approx_X})$ in the opposite direction. We say that a function $f : X \to Y$ between two sets X and Y is surjective, if $\Pi x : X.\, \|\Sigma y : Y.\, f\,x \equiv y\|$. The function θ is generally not surjective. But it can be proven surjective assuming countable choice: for all sets Y and relations P between $\mathbb{N}$ and Y, we assume $\Pi n : \mathbb{N}.\, \|\Sigma y : Y.\, P\,n\,y\| \to \|\Sigma f : \mathbb{N} \to Y.\, \Pi n : \mathbb{N}.\, P\,n\,(f\,n)\|$. To actually have an inverse ψ for θ we need more. In addition to countable choice, we need the quotient X/R to be weakly effective in the sense that $\Pi x, x' : X.\, \mathsf{abs}\,x \equiv \mathsf{abs}\,x' \to \|xRx'\|$. This can be proved by additionally assuming that equivalent propositions are equal: if two sets X, Y are propositions, then $X \leftrightarrow Y \to X \equiv Y$.

Assuming countable choice and equality of equivalent propositions (in addition to the assumptions of equality of extensionally equal functions and equality of (strongly) bisimilar coinductive data), multiplication $\mu_X : \hat{\mathsf{D}}\,(\hat{\mathsf{D}}\,X) \to \hat{\mathsf{D}}\,X$ can be defined and satisfies the monad laws.

One should be careful to not assume too much, as it is easy to arrive at provability of nonconstructive principles such as excluded middle. For example, assuming effectiveness for all quotients together with equality of equivalent propositions gives excluded middle. Also, it may be tempting to postulate surjectivity of the canonical function $(X \to Y)/(X \to R) \to X \to Y/R$ which is there for all X, Y and R. But this is logically equivalent to weak existence of a section for abs for all quotients: for all X and R, we have $\|\Sigma f : X/R \to X.\, \Pi\, q : X/R.\, \mathsf{abs}\,(f\,q) \equiv q\|$. And that in turn is logically equivalent to the full axiom of choice: for all sets X, Y and relations P between X and Y, it holds that $\Pi x : X.\, \|\Sigma y : Y.\, P\,x\,y\| \to \|\Sigma f : X \to Y.\, \Pi x : X.\, P\,x\,(f\,x)\|$.

A yet different approach is not to quotient objects $\mathsf{D}\,X$ by the relations ${\approx_X}$ and aim at $\hat{\mathsf{D}}$ being a strong monad, but instead quotient homsets $\mathbf{Kl}(\mathsf{D})(X, Y)$, i.e., function spaces $X \to \mathsf{D}\,Y$, by the pointwise extension $X \to {\approx_Y}$. This gives an arrow on **Set** in the sense of Hughes.

We have formalized our development in Agda.

Acknowledgement This research was supported by the ERDF funded Estonian ICT national programme project "Coinduction", the Estonian Science Foundation grants No. 9219 and 9475 and the Estonian Ministry of Education and Research institutional research grant IUT33-13.

References

[1] N. Benton, A. Kennedy, C. Varming. Some domain theory and denotational semantics in Coq. In S. Berghofer, T. Nipkow, C. Urban, M. Wenzel, eds., *Proc. of 22nd Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLs 2009*, v. 5674 of *Lect. Notes in Comput. Sci.*, pp. 115–130. Springer, 2009.

[2] V. Capretta. General recursion via coinductive types. *Log. Methods in Comput. Sci.*, 1(2:1), 2005.

[3] M. Hofmann. *Extensional constructs in intensional type theory*, CPHS/BCS Distinguished Dissertations. Springer, 1997.

A Coq Formalization of a Sign Determination Algorithm in Real Algebraic Geometry

Cyril Cohen[1] and Mathieu Kohli[2]

[1] INRIA Sophia Antipolis – Méditerrannée, France
[2] ENS Cachan, France
cyril.cohen@inria.fr, mkohli@ens-cachan.fr

Abstract

One of the main problems in real algebraic geometry is root counting. Given a polynomial, we want to count the number of roots that satisfies constraints expressed as polynomial inequalities. A naive way is to compute an exponential number of time consuming quantities, called Tarski Queries. In this paper, we formalize an algorithm which allows to compute a linear number of them. We formally build a linear system, and we prove in Coq that the system is of small size. The proof that the solutions of this system are the numbers of roots satisfying the constraints is still ongoing.

Introduction

One of the main algorithms in real algebraic geometry is the cylindrical algebraic decomposition [4]. The purpose of such an algorithm is to give a precise representation of a partitioning of the space described by polynomial equations.

Many algorithms that are connected to the task of finding such a decomposition are described in a book by Basu, Pollack and Roy [1], which we use as a reference on algorithms in real algebraic geometry. In this talk we focus on the formalization one of the sign determination algorithms described in this book. The purpose of this algorithm is to count the number of roots of a given polynomial, that satisfy polynomial constraints. The standard way to compute such quantities is by linear combinations of Tarski Queries. The Tarski Query of two polynomials is an integer that can be computed using the coefficients of the numerator and denominator of the fraction.

Tarski queries and counting roots

Given a polynomial P we consider the roots of, and a polynomial Q which expresses sign constraints on the roots of P, the Tarski Query can be defined as follows:

$$\mathrm{TaQ}(P,Q) = \sum_{x \in \mathrm{roots}(P)} \mathrm{sign}(Q(x)).$$

Although there is an algorithmic way to compute this quantity using the coefficients of P and Q, we do not explain this here for the sake of readability. One can find detailed explanation on how to do so in [1, 3, 2].

The number of roots of P such that Q has sign $\sigma \in \{-1, 0, +1\}$ can be expressed as:

$$\mathrm{cnt}(P,Q,\sigma) = \sum_{\substack{x \in \mathrm{roots}(P) \\ \mathrm{sign}(Q(x))=\sigma}} 1.$$

Hence, one can express Tarski Queries in terms of the number of roots in the following way:

$$\begin{pmatrix}\mathrm{TaQ}(P,1) & \mathrm{TaQ}(P,Q) & \mathrm{TaQ}(P,Q^2)\end{pmatrix} = \begin{pmatrix}\mathrm{cnt}(P,Q,0) & \mathrm{cnt}(P,Q,+1) & \mathrm{cnt}(P,Q,-1)\end{pmatrix}\cdot\begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 1\\ 1 & -1 & 1\end{pmatrix}$$

More generally, given a family $\mathcal{Q}$ of n polynomials $Q_1,\ldots,Q_n$, and a family of sign constrains $\sigma\in\{-1,0,1\}^n$ one can express the Tarski Query $\mathrm{TaQ}(P,\mathcal{Q}^\alpha)$, in terms of $\mathrm{cnt}(P,\mathcal{Q},\sigma)$ using a linear transformation, where $\alpha\in\{0,1,2\}^n$,

$$\mathcal{Q}^\alpha = \prod_i Q_i^{\alpha_i},$$

and

$$\mathrm{cnt}(P,\mathcal{Q},\sigma) = \sum_{\substack{x\in\mathrm{roots}(P)\\ \forall i,\mathrm{sign}(Q_i(x))=\sigma_i}} 1.$$

We formalize the linear transformation as a matrix in Coq/SSReflect and we prove that the size of this matrix has the required bounds. The proof that this matrix is invertible is still in progress, but we influenced the writing of the paper proof in the process of writing the formal proof.

The formalization we describe here is intended as a replacement for a piece of code used in a previous work of the first author [3, 2]. Indeed, in this previous work, we use a naive sign determination algorithm were we have to invert a 3^n square matrix and compute 3^n Tarski Queries, where n is the number of polynomials. In fact we know in advance that at most r quantities of the form $\mathrm{cnt}(P,\mathcal{Q},\sigma)$ will be nonzero, since P has at most r roots, and we can build a square matrix of size r and compute at most $3n$ Tarski Queries.

The use of a proof assistant such as Coq in the formalization of these algorithms led us to write algorithms and conduct our proofs in a slightly different way than the one described in our reference [1]. The authors of the book then adopted some of our reformulations to modify their book, in order to make the description more precise or concise and to provide more detailed justifications.

Acknowledgment

We thank Marie-Françoise Roy for the numerous discussions we had together that enlightened us with her expertise of the subject.

References

[1] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*, v. 10 of *Algorithms and Computation in Mathematics*. Springer, 2006.

[2] C. Cohen. *Formalized Algebraic Numbers: Construction and First Order Theory*. PhD thesis, École polytechnique, 2012.

[3] C. Cohen and A. Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Log. Methods in Comput. Sci.*, 8(1:2), 2012.

[4] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proc. of 2nd GI Conf. on Automata Theory and Formal Languages*, v. 33 of *Lect. Notes in Comput. Sci.*, pp. 134–183. Springer, 1975.

A Formalized Checker for Size-Optimal Sorting Networks

Luís Cruz-Filipe and Peter Schneider-Kamp*

Department of Mathematics and Computer Science, University of Southern Denmark, Denmark
{lcf|petersk}@imada.sdu.dk

Sorting networks are hardware-oriented algorithms to sort a fixed number of inputs using a predetermined sequence of comparisons between them. They are built from a primitive operator—the *comparator*—, which reads the values on two channels, and interchanges them if necessary to guarantee that the smallest one is always on a predetermined channel. Comparisons between independent pairs of values can be performed in parallel, and the two main optimization problems one wants to address are: how many comparators do we need to sort n inputs (the *optimal size* problem); and how many computation steps do we need to sort n inputs (the *optimal depth* problem).

So far, most results on these two problems were obtained by special-purpose computer programs that performed exhaustive analysis of huge space states, using only a handful of mathematical results. In particular, in previous work [1] we proposed a generate-and-prune algorithm to show size optimality of sorting networks, and used it to confirm all known bounds for up to 8 inputs and to establish the previously unknown optimality of 25-comparator sorting networks on 9 inputs.

Our program includes a pruning step that takes two sequences of comparators and decides whether one of them can be ignored, by searching among all 9! permutations of 9 elements for one with a particular property. This expensive step, which often fails, has to be repeated billions of times throughout the whole run; therefore, the total execution required more than 10 years of computational time, or around one month on a state-of-the-art parallel cluster able to run 288 threads simultaneously. However, during execution we recorded the comparator sequences and permutations that allowed *successful* pruning steps, so that the whole run could later be independently validated in just a few hours of sequential computation by shortcutting the expensive search process. In this work we go one step further, and use this information to produce a formal proof of these results. Our proof is divided in three parts: (1) a formalization of the theory of size-optimal sorting networks, closely following the classical reference on the topic [4]; (2) a formalization of the generate-and-prune algorithm from [1], further optimized for performance, taking advantage of the information previously recorded; and (3) an extracted program, correct by design, that successfully replicated the previous executions of generate-and-prune.

The formalization of the theory. The formalization closely follows the presentation of sorting networks in [4], in particular when defining the concept of optimal size. It presents a number of problems regarding the handling of finite sets and combinatorial arguments involving permutations and cardinality, which we attempt to solve in a systematic way. As key results, we obtain operational characterizations of sorting networks.

However, it is not feasible to verify that a sequence of comparators on 9 inputs is a sorting network (the Coq process eventually crashes after a few hours of computation), which makes the execution of any generate-and-test algorithm in Coq unrealistic. This motivates the approach of using program extraction to confirm the optimality results we aim for.

*The authors were supported by the Danish Council for Independent Research, Natural Sciences. Computational resources were provided by the Danish Center for Scientific Computing.

The formalization of the checker. The original formalization of the checker closely followed the pseudo-code for generate-and-prune presented in [1], with one main difference: the expensive search step at the core of the pruning phase was replaced by information obtained from an oracle (a parameter in the formalization). This oracle is untrusted, so any data it provides is checked before it is used, and we formally prove soundness theorems stating that applying k steps of generate-and-prune on n channels will either: return a sorting network of the smallest possible size, if this is smaller than or equal to k; or return a negative answer, guaranteeing that no sorting network of size k or smaller exists.

However, the resulting extracted program had unfeasible execution times on 9 inputs, due to the huge size of the search space. Therefore, we optimized the underlying algorithm taking advantage of the fact that the oracle is *offline*: the data is freely available for preprocessing. That allowed us to reformulate the pruning step in linear, rather than quadratic, time (in the size of the search space). Simultaneously, we improved the data structures used by the checker in order to obtain performance gains in steps that are performed in constant time. After reproving the soundness properties, we were able to extract a program that ran in under a week on a modest processor.

The extracted program. The formalization uses an oracle as a parameter, which had to be implemented outside of Coq. This was achieved by means of a simple program that reads the (suitably preprocessed) files generated by the original proof and feeds them as arguments to the extracted checker. Since the checker does not trust the oracle, this information is verified, so we do not have to worry about showing that the implementation of the oracle is itself correct.

The major bottleneck in executing the checker was memory consumption, and we were forced to extract natural numbers into native types in order to bring the requirements down to a manageable level. However, as natural numbers are used only as identifiers for channels in comparator networks, this does not raise any issues regarding the correctness-by-design of the extracted program.

Results and future work. Running the extracted program, we formally confirmed all the values for size-optimal sorting networks up to 9 inputs, as described in [2, 3]. The formalization is available at `http://imada.sdu.dk/~petersk/sn/`. During this process, we identified a few minor mistakes in previously published proofs, the fixing of which required adjusting some definitions. We would like to extend this work with the theory of depth-optimal sorting networks, which is substantially different from the one currently formalized, as well as with a formal proof of a theoretical result regarding size-optimality [5].

References

[1] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *Proc. of ICTAI 2014*, pp. 186–193. IEEE, 2014.

[2] L. Cruz-Filipe and P. Schneider-Kamp. Formalizing size-optimal sorting networks: Extracting a certified proof checker. Submitted. arXiv:1502.05209, 2015.

[3] L. Cruz-Filipe and P. Schneider-Kamp. Optimizing a certified proof checker for a large-scale computer-generated proof. In *Proc. of CICM 2015*, *Lect. Notes in Comput. Sci.*, Springer, to appear.

[4] D. Knuth. *The Art of Computer Programming, Volume III.* Addison-Wesley, 1973.

[5] D. van Voorhis. Toward a lower bound for sorting networks. In R. Miller and J. Thatcher, eds., *Complexity of Computer Computations*, *IBM Res. Symp. Series*, pp. 119–129. Plenum Press, 1972.

A Typed λ-Calculus with Call-by-Name and Call-by-Value Iteration

Herman Geuvers

Radboud Universiteit Nijmegen and Technische Universiteit Eindhoven, The Netherlands
h.geuvers@cs.ru.nl

We define a typed λ-calculus $\lambda 2\mu$It where data-types are represented as *Scott data types* (as opposed to the more well-known *Church data-types*), and where computation is done via continuations. Scott data types don't have any recursion built in, so we add iteration schemes to define functions by well-founded structural recursion. We have two separate schemes: one for *call-by-value iteration* and one for *call-by-name iteration*. The advantage is that we can control the computation behavior of functions via their definition, and not via the evaluation relation we choose: evaluation is done via weak head reduction. We show that our calculus is strongly normalizing and we define a notion of model for the calculus. Our calculus has polymorphism and recursive types, so the strong normalization proof is somewhat like the one of Mendler [3]. However, we have some additional features, like iterators over Scott data types, so we cannot just reuse the proof of Mendler. We first define a saturated-sets model, in the style of Krivine [2], and then show that the SN proof is almost an instance of it. We also show how we can switch between call-by-value to call-by-name, via a kind of double negation translation and we show how to define storage operators (in the sense of Krivine).

We now focus on the natural numbers to explain and motivate our system $\lambda 2\mu$It, polymorphic lambda calculus with positive recursive types and CBN and CBV iterators. The Scott numerals are defined as follows: $\underline{0} := \lambda x\, f.x$, $\underline{1} := \lambda x\, f.f\, \underline{0}$, $\underline{2} := \lambda x\, f.f\, \underline{1}$ and in general $\underline{p+1} := \lambda x\, f.f\, \underline{p}$ with the successor function defined by $\underline{S} := \lambda n.\lambda x\, f.f\, n$. The Scott numerals have *case* as a basis: the numerals are *case distinctors*: $\underline{n}\, q\, t = q$ if $n = 0$ and $\underline{n}\, q\, t = t\underline{m}$ if $n = m + 1$. An advantage is that the predecessor can immediately be defined: $\mathrm{pred} := \lambda n.n\, \underline{0}\, (\lambda x.x)$. On the other hand, one has to get "recursion" from somewhere else (e.g. in the untyped λ-calculus by using a fixed point-combinator). We also have the more well-known Church numerals, which are *iterators*: $\overline{n}\, x\, f$ iterates the function f on x, for n-times.

To type Scott numerals we need $\mathsf{N} = A \to (\mathsf{N} \to A) \to A$. In $\lambda 2$, we cannot do this, unless we extend it with (positive) recursive types [1]. Then one still does not get any well-founded recursion "for free", so we still have to add iterators. In $\lambda 2\mu$It, we can define the type of natural number as follows: $\mathsf{N} = \forall X.X \to (\mathsf{N} \to X) \to X$. The constructors for N are (with explicit type information in the λ-abstractions for clarity): $\mathsf{zero} := \lambda z : X.\lambda s : \mathsf{N} \to X.z : \mathsf{N}$, $\mathsf{suc} := \lambda n : \mathsf{N}.\lambda z : X.\lambda s : \mathsf{N} \to X.s\, n : \mathsf{N} \to \mathsf{N}$.

Let B be some data type and let A be a type. $\neg_A \mathsf{B}$ denotes $\mathsf{B} \to A$ and similarly, $\neg_A \neg_A \mathsf{B}$ denotes $(\mathsf{B} \to A) \to A$. The call-by-value iteration scheme for N is

$$\text{(It-CBV)} \quad \frac{f_1 : \neg_A \neg_A \mathsf{B} \quad f_2 : \mathsf{B} \to \neg_A \neg_A \mathsf{B} \quad c : \neg_A \mathsf{B} \quad n : \mathsf{N}}{\mathsf{Itcbv}\, f_1\, f_2\, c\, n : A}$$

With reduction rule

$$\mathsf{Itcbv}\, f_1\, f_2\, c\, n \quad \to_v \quad n\, (f_1\, c)\, (\lambda x : \mathsf{N}.\mathsf{Itcbv}\, f_1\, f_2\, (\lambda y : \mathsf{B}.f_2\, y\, c)\, x)$$

We define addition in call-by-value style, $\mathsf{AddCBV} : \forall X.\mathsf{N} \to \mathsf{N} \to \neg_X\neg_X\mathsf{N}$, as follows,

$$\mathsf{AddCBV} \quad := \quad \lambda x\, y : \mathsf{N}.\lambda c : \neg_X\mathsf{N}.\mathsf{Itcbv}\, \hat{y}\, \widehat{\mathsf{suc}}\, c\, x$$

with $\hat{y} := \lambda c.c\, y$, $\widehat{\mathsf{suc}} := \lambda n\, c.c(\mathsf{suc}\, n)$. Then we have the nice property:

$$\mathsf{AddCBV}\, \underline{n}\, \underline{m}\, c \longrightarrow\!\!\!\!\!\rightarrow^{wh} c\, \underline{n+m}.$$

For call-by-name, we let B be a data type with

$$\mathsf{B} = \forall X.(\Phi^1(\mathsf{B}) \to X) \to \ldots \to (\Phi^m(\mathsf{B}) \to X) \to X.$$

The call-by-name iterator for N is (for A an arbitrary type):

$$\text{(It-CBN)} \quad \frac{f_1 : \mathsf{B} \quad f_2 : \mathsf{B} \to \mathsf{B} \quad c_1 : \Phi^1(\mathsf{B}) \to A \ldots c_m : \Phi^m(\mathsf{B}) \to A \quad n : \mathsf{N}}{\mathsf{Itcbn}\, f_1\, f_2\, n\, c_1 \ldots c_m : A}$$

With reduction rule:

$$\mathsf{Itcbn}\, f_1\, f_2\, n\, \overline{c} \quad \to_n \quad n\, (f_1\, \overline{c})\, (\lambda x : \mathsf{N}.f_2\, (\lambda \overline{z}.\mathsf{Itcbn}\, f_1\, f_2\, x\, \overline{z})\overline{c})$$

We now define addition in call-by-name style, $\mathsf{AddCBN} : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$, as follows.

$$\mathsf{AddCBN} \quad := \quad \lambda x\, y : \mathsf{N}.\lambda c_1 : X.\lambda c_2 : \mathsf{N} \to X.\mathsf{Itcbn}\, y\, \mathsf{suc}\, x\, c_1\, c_2$$

Then $\mathsf{AddCBN}\, \underline{n}\underline{m}$ and $\underline{n+m}$ are observationally equal: $\mathsf{AddCBN}\, \underline{n}\, y =_{\beta\eta vn} \mathsf{suc}^n(y)$.

Storage operators have been introduced by Krivine to mimic call-by-value using call-by-name evaluation. In $\lambda 2\mu$It, a *storage operator* for data type D is a term $\mathsf{Stor} : \mathsf{D} \to \forall X.\neg_X\neg_X\mathsf{D}$ satisfying, for all M with $M =_{\beta vn} \underline{d}$,

$$\mathsf{Stor}\, M\, f \longrightarrow\!\!\!\!\!\rightarrow^{wh} f\, \underline{d}.$$

As the property also holds for a variable f and the evaluation $\to^{wh}$ is deterministic, a storage operator must *first* compute M to a normal form and then pass it to f (for any f). Define the two terms $\mathsf{Stor}_\mathsf{N} : \mathsf{N} \to \forall X.\neg_X\neg_X\mathsf{N}$ and $\mathsf{Unstor}_\mathsf{N} : (\forall X.\neg_X\neg_X\mathsf{N}) \to \mathsf{N}$ as follows.

$$\begin{aligned} \mathsf{Stor}_\mathsf{N} &:= \lambda n\lambda f.\mathsf{Itcbv}\, \widehat{\mathsf{zero}}\, (\lambda m\, b.b(\mathsf{suc}\, m))\, f\, n \\ \mathsf{Unstor}_\mathsf{N} &:= \lambda f.\lambda z\, s.f\, (\lambda n.n\, z\, s) \end{aligned}$$

with $\widehat{\mathsf{zero}} := \lambda c.c\, \mathsf{zero}$. Then the term Stor_N is a storage operator for N and $\mathsf{Unstor}_\mathsf{N}(\mathsf{Stor}_\mathsf{N}\, \underline{n}) =_{\beta vn} \underline{n}$ for all $n \in \mathbf{N}$.

Scott numerals have recently been studied intensively in the Implicit Computational Complexity setting, e.g. by Baillot, Brunel and Terui, It is future work to compare our approach with this line of research.

References

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

[2] J.-L. Krivine. Classical logic, storage operators and second-order lambda-calculus. *Ann. of Pure and Appl. Log.*, 68(1):53–78, 1994.

[3] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. of Pure and Appl. Log.*, 51(1–2):159–172, 1991.

Two-Dimensional Proof-Relevant Parametricity

Neil Ghani, Fredrik Nordvall Forsberg and Federico Orsanigo

Dept. of Comput. and Inform. Sci., University of Strathclyde, United Kingdom
{neil.ghani|fredrik.nordvall-forsberg|federico.orsanigo}@strath.ac.uk

Relational parametricity is a fundamental concept within theoretical computer science and the foundations of programming languages, introduced by John Reynolds [6]. His fundamental insight was that types can be interpreted not just as functors on the category of sets, but also as equality preserving functors on the category of relations. This gives rise to a model where polymorphic functions are uniform in a suitable sense; this can be used to establish e.g. representation independence, equivalences between programs, or deriving useful theorems about programs from their type alone [7].

The relations Reynolds considered were proof-irrelevant, which from a type theoretic perspective is a little limited. As a result, one might like to extend his work to deal with proof-relevant, i.e. set-valued relations. However naive attempts to do this fail: the fundamental property of equality preservation cannot be established. Our insight is that just as one uses parametricity to restrict definable elements of a type, one can use parametricity to restrict definable proofs to ensure equality preservation in the proof-relevant setting.

Proof-relevant logical relations also appear in Benton, Hofmann and Nigam's recent work [1], but for unary relations, and in a setting without $\forall$-types. Hence they do not need to consider equality preservation. In our work, we consider proof-relevant binary relations. A proof-relevant relation R over two sets A and B consists of a map $R\colon A \times B \to \mathsf{Set}$ which, for every two elements $(a, b) \in A \times B$, associates a set of proofs that they are related. On top of this, we have a new proof-irrelevant layer: a 2-dimensional relation Q is defined over four sets and four binary proof-relevant relations. It can be thought of as over a square, where the vertices are sets and the edges are the binary proof-relevant relations:

$$\begin{array}{ccc} A & \overset{R_1}{\leftrightarrows} & B \\ {\scriptstyle R_2}\uparrow\downarrow & Q & \uparrow\downarrow{\scriptstyle R_4} \\ C & \underset{R_3}{\leftrightarrows} & D. \end{array}$$

The 2-dimensional relation Q is defined to be a proof-irrelevant relation over proofs agreeing on the vertices: $Q(a, b, c, d) \subset R_1(a, b) \times R_2(a, c) \times R_3(c, d) \times R_4(b, d)$, where $(a, b, c, d) \in A \times B \times C \times D$.

Equality preservation, i.e. the Identity Extension Property, is fundamental for parametricity. The equality relation on a set gives a functor Eq from sets to relations, where $\mathsf{Eq}(a, b) = \{*|a = b\}$. Clearly, there is no unique way to define an corresponding map from binary relations to 2-dimensional relations. In the relational interpretation of $\forall$-types we instead use three different "equality" maps, represented by the following squares:

$$\begin{array}{ccc} A & = & A \\ \uparrow\downarrow & \mathsf{Eq}_1 & \uparrow\downarrow \\ B & = & B \end{array} \qquad \begin{array}{ccc} A & \leftrightarrows & B \\ \| & \mathsf{Eq}_2 & \| \\ A & \leftrightarrows & B \end{array} \qquad \begin{array}{ccc} A & \leftrightarrows & B \\ \uparrow\downarrow & \Gamma_1 & \| \\ B & = & B. \end{array}$$

Note that the structure involved is highly symmetric. In fact, Eq_2 arises from Eq_1 by reflection along the diagonal. Similarly, the same reflection along the diagonal applied to Γ_1 gives another

notion of "equality". We can easily define a group action on squares and binary proof-relevant relations, and it looks like the interpretation of types are invariant under this action, although we are in the middle of the proof. If this holds, it would simplify the presentation of the interpretation of $\forall$-types and make it easier to work with them.

So far we proved that each layer of sets, proof-relevant relations and 2-dimensional relations is fibred over the layer below. This structure gives a parametric 2-dimensional proof-relevant model for system F, where types are interpreted as equality-preserving lifted functors and terms as lifted natural transformations. In this interpretation, the Identity Extension Lemma and the Abstraction Theorem hold. In particular we need to generalize the statement of the Identity Extension Lemma, since we now have more kinds of "equalities". The (bi)fibrational structure of the model can be used to derive the standard consequences of parametricity [5]. For this, we need to generalize also the Graph Lemma, which is fundamental in the proofs of existence of initial algebras and terminal coalgebras. We are checking the details of the latter proofs, and the proof that parametricity implies (di)naturality.

Interestingly, these 2-dimensional relations have clear higher dimensional analogues where $(n+1)$-relations are fibred over a n-cube of n-relations [4]. Thus the story of proof relevant logical relations quickly expands into one of higher dimensional structures similar to the cubical sets which arises in Homotopy Type Theory [3]. Of course, there are also connections to Bernardy and Moulin's work on internal parametricity [2].

References

[1] N. Benton, M. Hofmann, and V. Nigam. Proof-relevant logical relations for name generation. In *Proc. of 11th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2013*, v. 7941 of *Lect. Notes in Comput. Sci.*, pp. 48–60. Springer, 2013.

[2] J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *Proc. of 27th Ann. IEEE Symp. on Logic in Computer Science, LICS '12*, pp. 135–144. IEEE CS, 2012.

[3] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In *Proc. of 19th Int. Conf. on Types for Proofs and Programs, TYPES 2013*, v. 26 of *Leibniz Int. Proc. in Inform.*, pp. 107–128. Dagstuhl, 2014.

[4] M. Grandis. The role of symmetries in cubical sets and cubical categories (on weak cubical categories, I). *Cah. Topol. Géom. Diff. Catég.*, 50:102–143, 2009.

[5] P. Johann, N. Ghani, F. Nordvall Forsberg, F. Orsanigo, and T. Revell. Bifibrational functorial semantics of parametric polymorphism. Draft, 2015. `https://personal.cis.strath.ac.uk/federico.orsanigo/bifibParam.pdf`

[6] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, ed., *Proc. of IFIP 9th World Computer Congress, Information Processing '93*, pp. 513-523. North-Holland ,1983.

[7] P. Wadler. Theorems for free! In *Proc. of 4th Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA '89*, pp. 347–359. ACM, 1989.

Intersection Types Fit Well with Resource Control

Silvia Ghilezan[1], Jelena Ivetić[1], Pierre Lescanne[2] and Silvia Likavec[3]

[1] Faculty of Technical Sciences, University of Novi Sad, Serbia
[2] University of Lyon, École Normal Supérieure de Lyon, France
[3] Dipartimento di Informatica, Università di Torino, Italy
{gsilvia|jelenaivetic}@uns.ac.rs, pierre.lescanne@ens-lyon.fr, likavec@di.unito.it

The notion of resource awareness and control has gained an important role both in theoretical and practical domains: in logic and lambda calculus as well as in programming languages and compiler design. The idea to control the use of formulae is present in Gentzen's sequent calculus' structural rules ([6]), whereas the idea to control the use of variables can be traced back to Church's λI-calculus ([5]). The augmented ability to control the number and order of uses of operations and objects has a wide range of applications ([9]) which enables, among others, compiler optimisations and memory management .

In this paper, we investigate the control of resources in the λ-calculus. We propose the $\lambda_{\circledR}$-calculus, a λ-calculus enriched with resource control operators. The explicit control of resources is enabled by the presence of *erasure* and *duplication* operators, which correspond to thinning and contraction rules in the type assignment system. Erasure is the operation that indicates that a variable is not present in the term anymore, whereas duplication indicates that a variable will have two occurrences in the term which receive specific names to preserve the "linearity" of the term. Indeed, in order to control all resources, in the spirit of the λI-calculus, void lambda abstractions are not acceptable, so in order to have $\lambda x.M$ well-formed the variable x has to occur in M. But if x is not used in the term M, one must perform an *erasure* by using the expression $x \odot M$. In this way, the term M does not contain the variable x, but the term $x \odot M$ does. Similarly, a variable should not occur twice. If nevertheless, we want to have two positions for the same variable, we have to duplicate it explicitly, using fresh names. This is done by using the operator $x <^{x_1}_{x_2} M$, called *duplication* which creates two fresh variables x_1 and x_2.

Resource control calculus We first introduce the syntax and reduction rules of the $\lambda_{\circledR}$-calculus. Explicit control of erasure and duplication leads to decomposition of reduction steps into more atomic steps. Since erasing and duplicating of (sub)terms essentially changes the structure of a program, it is important to see how this mechanism really works and to be able to control this part of computation. We chose a direct approach to term calculi rather than taking a more common path through linear logic [1, 3].

Although the design of our calculus has been motivated by theoretical considerations, it may have practical implications as well. Indeed, in the description of compilers by rules with binders in [8], the implementation of substitutions of linear variables by inlining is simple and efficient when substitution of duplicated variables requires the cumbersome and time consuming mechanism of pointers and it is therefore important to tightly control duplication. On the other hand, a precise control of erasing does not require a garbage collector and prevents memory leaking.

Intersection types and strong normalisation Intersection types were introduced to overcome the limitations of the simple type discipline (e.g. [2]), and they became a powerful tool for

characterising strong normalisation in different term calculi. Intersection types in the presence of resource control operators were first introduced in [7], where two systems with idempotent intersection were proposed. Later, non-idempotent intersection types for contraction and weakening are treated in [4]. In this paper, we treat a general form of intersection without any assumptions about idempotence. As a consequence, our intersection type system can be considered both as idempotent or as non-idempotent, both options having their benefits depending on the motivation.

We propose an intersection type assignment system $\lambda_{®}\cap$ that integrates intersection into logical rules, thus preserving syntax-directedness of the system. We assign a restricted form of intersection types to terms, namely strict types. Intersection types fit naturally with resource control. Indeed, the control allows us to consider three roles of variables: variables as placeholders, variables to be duplicated and variables to be erased. For each kind of a variable, there is a kind of type associated to it, namely a strict type for a placeholder, an intersection type for a variable to-be-duplicated, and a specific type constant $\top$ for an erased variable.

By the means of the introduced intersection type assignment system $\lambda_{®}\cap$, we manage to completely characterise strong normalisation in $\lambda_{®}$, i.e. we prove that terms in the $\lambda_{®}$-calculus enjoy strong normalisation if and only if they are typeable in $\lambda_{®}\cap$. First, we prove that all strongly normalising terms are typeable in the $\lambda_{®}$-calculus by using typeability of normal forms and redex subject expansion. We then prove that terms typeable in $\lambda_{®}$-calculus are strongly normalising by adapting the reducibility method for explicit resource control operators.

The presented work is significantly improved in comparison with [7], and it has been under revision for publication.

References

[1] S. Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1–2):3–57, 1993.

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

[3] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, eds., *1st Int. Conf. on Typed Lambda Calculi and Appl., TLCA '93*, v. 664 of *Lect. Notes in Comput. Sci.*, pp. 75–90. Springer, 1993.

[4] A. Bernadet and S. Lengrand. Non-idempotent intersection types and strong normalisation. *Log. Methods in Comput. Sci.*, 9(4), 2013.

[5] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[6] G. Gentzen. Unterschungen über das logische Schliessen. *Math Z.*, 39:176–210, 1935. Reprinted in M. Szabo, ed., *Collected papers of Gerhard Gentzen*, pp. 68–131. North-Holland, 1969.

[7] S. Ghilezan, J. Ivetić, P. Lescanne, and S. Likavec. Intersection types for the resource control lambda calculi. In A. Cerone and P. Pihlajasaari, eds., *Proc. of 8th Int. Coll. on Theoretical Aspects of Computing, ICTAC 2011*, v. 6916 of *Lect. Notes in Comput. Sci.*, pp. 116–134. Springer, 2011.

[8] K. H. Rose. CRSX - combinatory reduction systems with extensions. In M. Schmidt-Schauß, ed., *Proc. of 22nd Int. Conf. on Rewriting Techniques and Applications, RTA'11*, v. 10 of *Leibniz Int. Proc. in Inform.*, pp. 81–90. Dagstuhl, 2011.

[9] D. Walker. Substructural type systems. In B. Pierce, ed., *Advanced Topics in Types and Programming Languages*, pp. 3–44. MIT Press, 2005.

Colored Intersection Types: A Bridge between Higher-Order Model-Checking and Linear Logic

Charles Grellois and Paul-André Melliès

Laboratoires LIAFA & PPS, CNRS & Université Paris Diderot, France
{grellois,mellies}@pps.univ-paris-diderot.fr

The model-checking problem for higher-order recursive programs, expressed as *higher-order recursion schemes* (HORS), and where properties are specified in *monadic second-order logic* (MSO) has received much attention since it was proven decidable by Ong ten years ago. Every HORS may be understood as a simply-typed λ-term $\mathcal{G}$ with fixpoint operators Y whose free variables $a, b, c \ldots \in \Sigma$ are of order at most one. Following the principles of a Church encoding, these variables provide the tree constructors of a ranked alphabet Σ, so that the normalization of the recursion scheme $\mathcal{G}$ produces a typically infinite *value tree* $\langle\mathcal{G}\rangle$ over this ranked alphabet.

In order to check whether a given MSO formula φ holds at the root of such a value tree $\langle\mathcal{G}\rangle$, a convenient and traditional approach is to run an equivalent automaton $\mathcal{A}_\varphi$ over it. In the specific case of MSO logic, the corresponding notion of automaton is provided by *alternating parity tree automata* (APT), a kind of non-deterministic top-down tree automaton enriched with *alternation* and *coloring*. Every run of such an automaton may be understood as a syntactic proof-search of the validity of the formula φ over the value tree $\langle\mathcal{G}\rangle$. A typical transition over a binary symbol $a \in \Sigma$ is of the following shape:

$$\delta(q_0, a) \quad = \quad (2, q_2) \quad \vee \quad ((1, q_1) \wedge (1, q_2) \wedge (2, q_0))$$

When reading the symbol a in the state q_0, the automaton $\mathcal{A}_\varphi$ can either (1) drop the left subtree of a, and explore the right subtree with state q_2, or (2) explore twice the left subtree of a in parallel, once with state q_1 and the other time with state q_2, and explore the right subtree of a with state q_0. Kobayashi observed that the transitions of an alternating tree automaton $\mathcal{A}$ can be reflected by giving to the symbol a the following refined intersection type:

$$a \ : \ (\emptyset \to q_2 \to q_0) \wedge ((q_1 \wedge q_2) \to q_0 \to q_0) \tag{1}$$

Using intersection types in this way, Kobayashi constructs a type system where a higher-order recursion scheme $\mathcal{G}$ is typed by a state q_0 of the automaton $\mathcal{A}$ iff its value tree $\langle\mathcal{G}\rangle$ is recognized from that state q_0. In order to recover the full expressive power of MSO logic, one needs to adapt this correspondence theorem to alternating *parity* automata (APT), and thus to integrate colors in the intersection type system. Recall that every state q of such an APT is assigned a color $\Omega(q) \in \mathbb{N}$. This additional information is devised so that a run-tree of the APT $\mathcal{A}_\varphi$ over the value tree $\langle\mathcal{G}\rangle$ proves the validity of the associated MSO formula φ iff, for every infinite branch of the run-tree, the greatest color encountered infinitely often is even. Kobayashi and Ong extended the original intersection type system in order to integrate this extra coloring information.

In a series of recent papers [6, 7], we establish a tight and somewhat unexpected connection between higher-order model-checking and linear logic, starting from a modal reformulation of Kobayashi and Ong's work. In particular, we show that their original type system can be slightly altered (and in fact improved) in order to disclose the modal nature of colors, and its connection to the exponential modality of linear logic. In our modal reformulation, the refinement type (1) associated to the transition of an APT may be colored (or modalised) in

the following way:

$$a \ : \ (\emptyset \to \Box_{c_2} \ q_2 \to q_0) \wedge ((\Box_{c_1} \ q_1 \wedge \Box_{c_2} \ q_2) \to \Box_{c_0} \ q_0 \to q_0) \tag{2}$$

where $\Box_c$ describes a family of modal operators, indexed by colors $c \in \mathbb{N}$. The connection of intersection types with linear logic comes from the linear decomposition of the intuitionistic arrow

$$A \Rightarrow B \quad = \quad !A \multimap B$$

which regards a program of type $A \Rightarrow B$ as a program of type $!A \multimap B$ which thus uses its input $!A$ only once in order to compute its output B ; but where the exponential modality "!" enables at the same time the program to discard or to duplicate this single input $!A$. In the relational semantics of linear logic, the exponential modality ! is interpreted as a *finite multiset* construction, so that the model keeps track of the number of times an argument is called by the function. The relational semantics is called *quantitative* for that reason. We translate in [5] the intersection type system originally devised by Kobayashi (restricted to the simply-typed λ-calculus) into an equivalent intersection type system where intersection is non-idempotent. Adapting a correspondence developped by Bucciarelli and Ehrhard [1, 2] between indexed linear logic and the relational semantics of linear logic, we establish that the resulting intersection type system computes the relational semantics of simply-typed λ-terms. At this stage, there remains to extend the correspondence to the simply-typed λ-calculus with a fixpoint operator Y. One conceptual difficulty is that the traditional interpretation of $!A$ in the relational semantics of linear logic is biased towards an inductive (rather than coinductive) interpretation of the fixpoint operator Y. Technically speaking, this comes from the fact that the multisets in $!A$ are *finite*. For that reason, we develop an alternative relational semantics of linear logic where the exponential modality noted $A \mapsto \not!\, A$ is interpreted as the set $\mathcal{M}_{\leq \omega}(A)$ of *finite-or-countable* multisets of elements of A, see [6] for details. This alternative and "infinitary" relational interpretation of linear logic enables us to establish a clean correspondence between (1) the coinductive intersection type system originally constructed by Kobayashi (2) the run-trees of an alternating tree automaton with coinductive acceptance condition (3) our "infinitary" variant of the traditional relational semantics of linear logic. Put all together, these results provide a semantic account of higher-order model-checking where the acceptance condition of the underlying alternating tree automaton $\mathcal{A}$ is restricted however to the purely coinductive case.

At this stage, there thus remained to capture the full power of the MSO logic. To that purpose, we incorporated the family $\Box_c$ of modal operators mentioned earlier to our infinitary relational semantics of linear logic. The key idea is that this extra coloring information living at the level of the intersection type system reduces once reformulated at the level of the relational semantics into a very simple and elementary comonad, defined as follows:

$$\Box \, A \quad = \quad Col \times A \quad = \quad \&_{c \in Col} \ A$$

where $Col \subseteq \mathbb{N}$ typically denotes the finite set of colors appearing in the alternating parity tree automaton $\mathcal{A}_\varphi$ associated to the MSO-formula φ. The existence of a distributive law $\lambda : \not! \circ \Box \Rightarrow \Box \circ \not!$ enables us to compose the comonad $\Box$ with the exponential modality $\not!$ in the original relational semantics, in order to obtain a new and "colored" exponential modality $A \mapsto \not! \, \Box \, A$. In the resulting infinitary and colored relational model, the colored intersection typing (2) has the following interpretation $[\![a]\!]$ of the symbol $a \in \Sigma$ as semantic counterpart:

$$[\![a]\!] \quad = \quad \{ \quad ([], ([(c_2, q_2)], q_0)) \quad , \quad ([(c_1, q_1), (c_2, q_2)], ([(c_0, q_0)], q_0)) \quad \} \tag{3}$$

Note that the interpretation of the symbol a of the alternating parity tree automaton $\mathcal{A}_\varphi$ is a

subset of the interpretation of $o \to o \to o$, where o is interpreted as the set Q in our colored relational semantics:

$$[\![a]\!] \quad \subseteq \quad \not{!}\,\square\, o \otimes \not{!}\,\square\, o \multimap o \quad = \quad \left(\mathcal{M}_{\leq\omega}(Col \times Q)\right)^2 \times Q$$

We then defined an inductive-coinductive fixpoint operator Y, based on the principles of alternating parity tree automata: the fixpoint operator iterates finitely in the scope of an odd color, and infinitely when the color is even, see [6] for details. This interpretation of the fixpoint operator Y based on parity may be also formulated at the level of intersection types: it corresponds in that setting to the introduction of a fixpoint rule, together with an appropriate notion of winning derivation tree formulated by the authors in [7]. Finally, we prove that a recursion scheme $\mathcal{G}$ produces a tree accepted from q by $\mathcal{A}_\varphi$ if and only if its colored relational semantics contains q – or alternatively, if and only if there is a winning derivation typing $\mathcal{G}$ with q.

This connection with linear logic leads us to a new proof of the decidability of the "selection problem" established by Carayol and Serre [3]. Our semantic proof of decidability [4] is based on the construction of a finitary and colored semantics of linear logic, adapted this time from the traditional *qualitative* semantics of linear logic based on prime-algebraic lattices and Scott-continuous functions between them — rather than from its alternative *quantitative* relational semantics. Interestingly, this qualitative semantics of linear logic corresponds to an intersection type system with subtyping, formulated in particular in the work by Terui [9]. It should be noted that the decidability of the "selection problem" implies in particular the decidability result for MSO formulas established by Ong [8] ten years ago. This decidability result gives a strong evidence of the conceptual as well as technical relevance of the connection which we have established and developed [4, 5, 6, 7] between higher-order model-checking and linear logic[1].

References

[1] A. Bucciarelli and T. Ehrhard. On phase semantics and denotational semantics in multiplicative-additive linear logic. *Ann. of Pure and Appl. Log.*, 102(3):247–282, 2000.

[2] A. Bucciarelli and T. Ehrhard. On phase semantics and denotational semantics: the exponentials. *Ann. of Pure and Appl. Log.*, 109(3):205–241, 2001.

[3] A. Carayol and O. Serre. Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In *Proc. of LICS 2012*, pp. 165–174. IEEE CS, 2012.

[4] C. Grellois and P. Melliès. Finitary semantics of linear logic and higher-order model-checking. arXiv:1502.05147, 2015. Submitted.

[5] C. Grellois and P. Melliès. Indexed linear logic and higher-order model checking. In *Proc. of ITRS 2014*, v. 177 of *Electron. Proc. in Theor. Comput. Sci.*, pp. 43–52. Open Publ. Assoc., 2015.

[6] C. Grellois and P. Melliès. An infinitary model of linear logic. In *Proc. of FoSSaCS 2015*, v. 9034 of *Lect. Notes in Comput. Sci.*, pp. 41–55. Springer, 2015.

[7] C. Grellois and P. Melliès. Relational semantics of linear logic and higher-order model-checking. arXiv:1501.04789, 2015. Submitted.

[8] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Proc. of LICS 2006*, pp. 81–90. IEEE CS, 2006.

[9] K. Terui. Semantic evaluation, intersection types and complexity of simply typed lambda calculus. In *Proc. of RTA 2012*, v. 15 of *Leibniz Int. Proc. in Inform.*, pp. 323–338. Dagstuhl, 2012.

[1] Although the papers mentioned here [4, 5, 6, 7] will be published this year, the truth is that it took us several years of work to carry out the connection between higher-order model-checking and linear logic described in this brief survey. The idea and the details of the connection were thus exposed in seminar talks and at various stages of development in the past three years.

Implementing Dependent Types Using Sequent Calculi

Daniel Gustafsson and Nicolas Guenot*

IT University of Copenhagen, Denmark
{dagu|ngue}@itu.dk

A critical issue concerning programming languages and proof assistants based on dependent type theory is the efficient implementation of reduction on *open terms*, where meta-variables can appear: this question is important not only for the execution of dependently typed programs, but already in the typechecker implementing the validation of such programs, through the use of the *conversion* rule. Following the Curry-Howard tradition, we are currently investigating the relation between the internal language of Agda [8] and a focused sequent calculus presentation of intuitionistic logic, the sequent calculus being naturally well-suited for typing open terms.

1 Internal Languages using Dependent Types

Different approaches can be used to implement a dependently typed language. For example, the proof terms used in the Coq proof assistant are really based on the natural deduction presentation of a higher-order intuitionistic logic, where matching is performed through a dedicated construct. However, in the Agda language, matching is part of an equational style of programming where the left-hand side of a definition can have a complex shape. As a result, the internal language used in Agda is not exactly a dependently typed pure λ-calculus, but a calculus containing the case-splitting tree of a definition involving matching.

A close look at the internal language of Agda reveals that it is much related to a presentation of intuitionistic logic in the sequent calculus. In particular, one can see the use of a definition as applying the *cut* rule, where one premise corresponds to the definition and the other to its use inside another term. Then, whenever the typechecker needs to compare two terms, it needs to reduce them by unfolding the definitions, which corresponds to a cut elimination process — note that all cuts cannot be eliminated, because of recursive definitions. At this point, it is important to observe that cut elimination in the sequent calculus, although equivalent to β-reduction in the standard λ-calculus, can be performed in many different ways [9, 3]. Moreover, important insights on cut elimination have been obtained through the analysis of linear logic [4] and its intuitionistic variant [2]. Based on the idea that more control can be gained over normalisation of terms in the sequent calculus, compared to natural deduction and the associated β rule, we propose an analysis of reduction in Agda using the tools of the sequent calculus to improve the efficiency of reduction, and therefore also the efficiency of typechecking.

2 Term Assignments for Focused Sequent Calculi

Although it offers a fine-grained representation of proofs, the sequent calculus LJ for intuitionistic logic is not well-suited for computation, in its basic, unrestricted form. Indeed, it lacks a canonical structure as found in natural deduction, that can guide the reduction process, but a well-behaved form can be recovered through the *focusing* technique [1, 7]. This strong structure is also important to compare proof terms, which is essential in a proof assistant — in the setting of

*Supported by grant 10-092309 from the Danish Council for Strategic Research to the *Demtech* project.

natural deduction, $\beta\eta$-normal forms are considered, while in the sequent calculus one should consider cut-free focused proofs. Moreover, focusing allows to consider only the fragment of proofs isomorphic to the standard λ-calculus, or to allow for proofs containing more *sharing*.

The internal language of Agda uses an application to list of arguments called *spines*, and is best compared to the $\overline{\lambda}$-calculus of Herbelin [5], the computational interpretation of LJT, a particular focused system related to call-by-name reduction in the λ-calculus. There are other possible choices in the design of a focused system, for example following a call-by-value approach, but LJT is an excellent starting point — in particular, because it is the basis for a system developped to handle dependent types in the sequent calculus [6]. Our claim is that the versatile structure of the sequent calculus, associated with the strong structure provided by focusing, makes it an ideal framework to study, from a proof-theoretical perspective, the implementation of dependently typed languages.

As an example, consider a focused system where $\vdash$ denotes the inversion phase and $\vDash$ denotes the focusing phase. One can handle case-splitting on natural numbers using the following rules:

$$\frac{\Gamma \vDash t : C}{\Gamma \vdash \mapsto t : C} \qquad \frac{\Gamma, \Gamma'\{\mathtt{zero}/x\} \vdash T : C\{\mathtt{zero}/x\} \quad \Gamma, n : \mathbb{N}, \Gamma'\{\mathtt{suc}\, n/x\} \vdash T' : C\{\mathtt{suc}\, n/x\}}{\Gamma, x : \mathbb{N}, \Gamma' \vdash \mathtt{split}\,(x, T, n.T') : C}$$

and this yields the following interpretation of an equational definition of addition, as it would be written in Agda:

```
add zero n = n
add (suc m') n = suc (add m' n)
```

$$\Rightarrow \; m : \mathbb{N}, n : \mathbb{N} \vdash \mathtt{split}\,(m, \mapsto n, m' \mapsto \mathtt{suc}\,(\mathtt{add}\,(m' :: n :: \varepsilon) :: \varepsilon))$$

From the basis of a sequent system adding dependent types to $\overline{\lambda}$, we can start the investigation of more complex focused systems, as obtained for example when considering the full LJF calculus of [7], of which the usual LJT and LJQ calculi are fragment. The goal of this move from the standard setting of natural deduction to the sequent calculus is to exploit the ability of the sequent calculus to capture different evaluation strategies, such as call-by-name and call-by-value, in a single, unifying framework, and therefore allowing to improve the efficiency of reduction.

References

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

[2] N. Benton, G. Bierman, M. Hyland, and V. de Paiva. A term calculus for intuitionistic linear logic. In *Proc. of TLCA '93*, v. 664 of Lect Notes in Comput. Sci., pp. 75–90. Springer, 1993.

[3] J. Gallier. Constructive logics part I: A tutorial on proof systems and typed λ-calculi. *Theor. Comput. Sci.*, 110(2):249–339, 1993.

[4] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50(1–2):1–102, 1987.

[5] H. Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Selected Papers from CSL '94*, v. 933 of *Lect. Notes in Comput. Sci.*, pp. 61–75, 1995.

[6] S. Lengrand, R. Dyckhoff, and J. McKinna. A focused sequent calculus framework for proof search in pure type systems. *Log. Methods in Comput. Sci.*, 7(1:6), 2011.

[7] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768, 2009.

[8] U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[9] A. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996.

A Proof with Side Effects of Gödel's Completeness Theorem Suitable for Semantic Normalisation

Hugo Herbelin

PPS, INRIA and University 7, France
hugo.herbelin@inria.fr

As revealed by T. Griffin [3], classical reasoning is related to control operators. Classical reasoning can be simulated within intuitionistic logic via a negative translation, the same way programming with control can be simulated in pure λ-calculus by reasoning within a continuation monad. Under this view, classical logic is "direct style" for proving intuitionistically within the target of a negative translation.

We shall consider a monotonic memory update effect and connect it to Kripke's forcing, seen as a dependent variant of the reader monad. We shall then interpret reasoning within the target of Kripke's forcing as indirect style for reasoning with a monotonic memory update effect (by monotonic update is meant that the memory can be updated only with a value which refines the previous value according to a given refinement order).

As an application, let us consider logical completeness of classical logic with respect to two-valued models and logical completeness of intuitionistic logic with respect to Kripke models and see the former as a direct-style formulation of the latter using monotonic memory update.

The core of the completeness proof proves $(\vDash_{\mathcal{M}} A) \leftrightarrow (\Gamma \vdash A)$ for the two-valued model defined on atoms by $\vDash_{\mathcal{M}} P \triangleq (\Gamma \vdash P)$, where Γ is a mutable variable denoting a context and updatable according to the context extension order.

It is formulated in an intuitionistic second-order arithmetic with delimited side-effects validating a specific principle characterising monotonic memory update. This arithmetic is intuitionistic in the sense that the disjunction and witness existence properties are retained.

The proof instructions for delimited monotonic update are `set` and `update` where T ranges hereditarily positive formulas, i.e. over $\forall$-$\rightarrow$-free formulas. The updatable variable is denoted by an ordinary term variable x.

$$\frac{\Gamma, [b : x \geq t] \vdash q : T(x) \qquad \Gamma \vdash r : \mathit{refl}\ \geq \quad \Gamma \vdash s : \mathit{trans}\ \geq \quad x \text{ fresh in } \Gamma \text{ and } T(t)}{\Gamma \vdash \mathtt{set}\, x := t\, \mathtt{as}\, b /_{(r,s)}\, \mathtt{in}\, q : T(t)}\ \text{SETEFF}$$

$$\frac{\Gamma, [b : x \geq t(x')] \vdash q : T(x) \quad \Gamma \vdash r : t(x') \geq x' \quad [x \geq u] \in \Gamma \text{ for some } u \quad x' \text{ fresh in } \Gamma}{\Gamma \vdash \mathtt{update}\, x := t(x)\, \mathtt{of}\, x'\, \mathtt{as}\, b\, \mathtt{by}\, r\, \mathtt{in}\, q : T(t(x))}\ \text{UPDATE}$$

The core of the completeness proof is defined for the negative fragment of classical logic by mutual induction using the "reify" $\downarrow$ and "reflect" $\uparrow$ functions used in the context of semantic normalisation (starting with [1]) and type-directed partial evaluation (e.g. [2]).

The core of the completeness proof is:

$$\begin{array}{lccl}
\uparrow_A & \Gamma \vdash A & \longrightarrow & \vDash_{\mathcal{M}} A \\
\uparrow_{P(\vec{t})} & g & \triangleq & g \\
\uparrow_{A \dot{\to} B} & g & \triangleq & m \mapsto \uparrow_B \dot{\mathtt{App}}_{\to}^{\Gamma,A,B}(g, \downarrow_A m) \\
\uparrow_{\dot{\forall} x\, A} & g & \triangleq & t \mapsto \uparrow_{A[t/x]} \dot{\mathtt{App}}_{\forall}^{\Gamma,x,A}(g, t)
\end{array}$$

$$\begin{array}{lccl}
\downarrow_A & \vDash_{\mathcal{M}} A & \longrightarrow & \Gamma \vdash A \\
\downarrow_{P(\vec{t})} & m & \triangleq & m \\
\downarrow_{A \dot{\to} B} & m & \triangleq & \dot{\mathtt{Abs}}_{\to}^{\Gamma,A}(\mathtt{update}\,\Gamma := (\Gamma, A)\,\mathtt{of}\,\Gamma_1\,\mathtt{as}\,b_A\,\mathtt{by}\,r_A\,\mathtt{in}\downarrow_B\ (m\,(\uparrow_A \dot{\mathtt{Ax}}^{\Gamma_1,A,,\Gamma}(b_A)))) \\
\downarrow_{\dot{\forall} x\, A} & m & \triangleq & \dot{\mathtt{Abs}}_{\forall}^{\Gamma,x,A}(\dot{y}, \downarrow_{A[z/x]}\ (m\,\dot{y}))
\end{array}$$

where $\dot{\mathtt{Abs}}_{\to}$, $\dot{\mathtt{App}}_{\to}$, $\dot{\mathtt{Abs}}_{\forall}$, $\dot{\mathtt{App}}_{\forall}$ and $\dot{\mathtt{Ax}}$ are the constructors of the logic while r_X proves $\Gamma \subset (\Gamma, X)$ and $\dot{y}$ is taken fresh in Γ.

Let $\mathit{Classic}(\mathcal{M}) \triangleq \forall A\,(\vDash_{\mathcal{M}} \dot{\neg}\dot{\neg} A \ \longrightarrow \vDash_{\mathcal{M}} A)$. The next step is to show that the model is classical:

$$\begin{array}{lcl}
\mathtt{classic} & : & \mathit{Classic}(\mathcal{M}) \\
\mathtt{classic} & \triangleq & A \mapsto m \mapsto \uparrow_A (\dot{\mathtt{Dn}}(\downarrow_{\dot{\neg}\dot{\neg} A} m))
\end{array}$$

with $\dot{\mathtt{Dn}}$ standing for double negation elimination.

The final statement is then:

$$\begin{array}{lcl}
\mathtt{compl}_A : & & \forall \mathcal{M} \forall \sigma\,(\mathit{Classic}(\mathcal{M}) \to \vDash_{\mathcal{M}} A) \longrightarrow \vdash A \\
\mathtt{compl}_A & \triangleq & \psi \mapsto \mathtt{set}\,\Gamma := \emptyset\,\mathtt{as}\,b/_{(r,s)}\,\mathtt{in}\downarrow_A (\psi\,\mathcal{M}\,\mathtt{id}\,\mathtt{classic})
\end{array}$$

where $\mathtt{id}$ is the empty substitution, $\emptyset$ is the empty context and r and s are proofs of reflexivity and transitivity of the inclusion $\subset$ of contexts.

The resulting proof of completeness does not require any enumeration of formulas. In our computational meta-language, it produces normal proofs of $\vdash A$ by replicating the structure of the initial proof of $\vDash A$, thus suitable for semantic normalisation.

The contents of the talk is covered by a work-in-progress paper.

References

[1] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proc. of 6th Ann. IEEE Symp. on Logic in Comput. Sci., LICS '91*, pp. 203–211. IEEE CS, 1991.

[2] O. Danvy. Type-directed partial evaluation. In *Conf. Record of 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '90*, pp. 242–257. ACM, 1996.

[3] T. G. Griffin. The formulae-as-types notion of control. In *Conf. Record of 17th Annual ACM Symp. on Principles of Programming Languages, POPL '90*, pp. 47–57. ACM, 1990.

A Lambek Calculus with Dependent Types

Zhaohui Luo*

Department of Computer Science, Royal Holloway, University of London, United Kingdom
zhaohui.luo@hotmail.co.uk

In this note, we discuss how to introduce dependent types into the Lambek calculus [7] (or, in general, an ordered calculus [17] and extensions such as those found in the studies of categorial grammars [12, 13]). One of the motivations to introduce dependent types in syntactical analysis is to facilitate a closer correspondence between syntax and semantics, especially when modern type theories are used in formal semantics [14, 9, 10, 2]. We then hope to establish a uniform basis for NL analysis: from automated syntactical analysis to logical reasoning in proof assistants based on type theories and formal semantics.

Dependent types have been used in various contexts in computational linguistics (see, for example, [14, 3, 11] among others). In [15] that formalises the Lambek calculus in (a proof assistant that implements) Martin-Löf's type theory, Ranta discussed the idea of introducing type dependency into directed types and gave an inspiring example to represent directed types of quantifiers, although the paper did not study a formal treatment of such an extension. De Groote et al [4] studied how to extend the underlying type system for ACGs by (intuitionistic) dependent product types so that type families can be used to represent syntactic categories indexed by linguistic features.

Combining resource sensitive types (such as those in linear logic [5]) with dependent types has been an interesting but difficult research topic and many researchers have worked on this, mainly with motivations to apply such calculi to programming and verification problems in computer science (see early work such as [1] and recent developments such as [6, 16]). However, how such a combination should be done is still widely open, on the one hand, and very much depends on the motivations in their applications, on the other.

We present a Lambek calculus with dependent types. Besides the type constructors in the Lambek calculus [7], we shall introduce directed types for dependent products (Π^r and Π^l) and dependent sums ($\Sigma^\sim$ and Σ°). Also considered is how to introduce type universes into the calculus. These are some of the type constructors in a modern type theory found useful in formal semantics and, therefore, their introduction at the syntactic level helps to facilitate a closer syntax-semantics tie as mentioned above. A focus of the current note is to formulate the rules for these types in ordered contexts so that their meanings are correctly captured.

1 The Lambek Calculus

Introducing dependent types into a Lambek calculus, we consider a calculus with contexts having two parts:

$$\Gamma;\ \Delta$$

where Γ is an intuitionistic context (whose variables can be used for any times in a term) and Δ is a Lambek context (or ordered context). Types may only depend on ordinary variables in Γ, but not on Lambek variables in Δ.[1] Since we are going to introduce dependent types in which

*Partially supported by grants from Royal Academy of Engineering and CAS/SAFEA International Partnership Program.

[1] This is a design choice of the current notes; it follows all the existing work (so far) on introducing dependent types into resource sensitive calculi.

$$(/\text{-F})\quad \frac{\Gamma;\ *\vdash A\ type \quad \Gamma;\ *\vdash B\ type}{\Gamma;\ *\vdash B/A\ type} \qquad (/\text{-I})\quad \frac{\Gamma;\ (\Delta, x{:}A)\vdash b : B \quad \Gamma;\ *\vdash B/A\ type}{\Gamma;\ \Delta\vdash /x{:}A.b : B/A}$$

$$(/\text{-E})\quad \frac{\Gamma;\ \Delta_1\vdash f : B/A \quad \Gamma;\ \Delta_2\vdash a : A}{\Gamma;\ (\Delta_1,\Delta_2)\vdash f\ a : B} \qquad (/\text{-C})\quad \frac{\Gamma;\ (\Delta_1, x{:}A)\vdash b : B \quad \Gamma;\ \Delta_2\vdash a : A}{\Gamma;\ (\Delta_1,\Delta_2)\vdash (/x{:}A.b)\ a = [a/x]b : B}$$

Figure 1: Rules for directed Lambek types B/A.

$$(\Pi^r\text{-F})\quad \frac{\Gamma;\ *\vdash A\ type \quad \Gamma, x{:}A;\ *\vdash B\ type}{\Gamma;\ *\vdash \Pi^r x{:}A.B\ type} \qquad (\Pi^r\text{-I})\quad \frac{\Gamma, x{:}A;\ \Delta\vdash b : B \quad \Gamma;\ *\vdash \Pi^r x{:}A.B\ type}{\Gamma;\ \Delta\vdash \lambda^r x{:}A.b : \Pi^r x{:}A.B}$$

$$(\Pi^r\text{-E})\quad \frac{\Gamma;\ \Delta\vdash f : \Pi^r x{:}A.B \quad \Gamma;\ *\vdash a : A}{\Gamma;\ \Delta\vdash app^r(f,a) : [a/x]B} \qquad (\Pi^r\text{-C})\quad \frac{\Gamma, x{:}A;\ \Delta\vdash b : B \quad \Gamma;\ *\vdash a : A \quad \Gamma;\ *\vdash \Pi^r x{:}A.B\ type}{\Gamma;\ \Delta\vdash app^r(\lambda^r x{:}A.b,\ a) = [a/x]b : [a/x]B}$$

Figure 2: Directed Π^r-types.

objects may occur, we need the following equality typing rule which says that computationally equal types have the same objects:

$$\frac{\Gamma;\Delta\vdash a : A \quad \Gamma;\ *\vdash A = B}{\Gamma;\Delta\vdash a : B}$$

Contexts of the above form obey the following validity rules where, in the last two rules, $x\notin FV(\Gamma,\Delta)$:

$$\frac{}{*;*\ valid} \qquad \frac{\Gamma;\Delta\ valid \quad \Gamma;\ *\vdash A\ type}{(\Gamma, x{:}A);\ \Delta\ valid} \qquad \frac{\Gamma;\Delta\ valid \quad \Gamma;\ *\vdash A\ type}{\Gamma;\ (\Delta, x{:}A)\ valid}$$

For variables, we have

$$\frac{\Gamma, x{:}A, \Gamma';\ *\ \ valid}{\Gamma, x{:}A, \Gamma';\ *\vdash x : A} \qquad \frac{\Gamma;\ y{:}A\ valid}{\Gamma;\ y{:}A\vdash y : A}$$

We can now present the directed types in the Lambek calculus. The rules for the directed types B/A are given in Figure 1. The rules for $A\setminus B$ are symmetric with term constructors such as $\setminus x{:}A.b$ and are omitted (so are the rules for ordered conjunctions).

2 Dependent Lambek Types

Directed Dependent Products. Dependent product types (Π-types) are split into directed dependent products (Π^r and Π^l). The rules for Π^r-types are given in Figure 2. The rules for Π^l-types, omitted here, are symmetric with term constructors $\lambda^l x{:}A.b$ and $app^l(a,f)$.

Type Universes. In type theory, a type universe is a type whose objects are (names of) types. For instance, common nouns can be considered to correspond to types (rather than predicates), as in formal semantics in modern type theories [14, 8]. In a similar fashion, we may introduce a universe CN of common nouns:

$$\frac{}{*;*\vdash CN\ type} \qquad \frac{\Gamma;\ *\vdash A : CN}{\Gamma;\ *\vdash T_{CN}(A)\ type}$$

$$(\Sigma^{\sim}\text{-F}) \quad \frac{\Gamma;\ *\vdash A\ type \quad \Gamma, x{:}A;\ *\vdash B\ type}{\Gamma;\ *\vdash \Sigma^{\sim}x{:}A.B\ type}$$

$$(\Sigma^{\sim}\text{-I}) \quad \frac{\Gamma;*\vdash a:A \quad \Gamma;\Delta\vdash b:[a/x]B}{\Gamma;\Delta\vdash pair(a,b):\Sigma^{\sim}x{:}A.B}$$

$$(\Sigma^{\sim}\text{-E}) \quad \frac{\Gamma;\Delta\vdash p:\Sigma^{\sim}x{:}A.B \quad \Gamma, x{:}A;\ \Delta', y{:}B\vdash e:C \quad \Gamma;\ *\vdash C\ type}{\Gamma;\ (\Delta,\Delta')\vdash \mathbf{let}\ pair(x,y)=p\ \mathbf{in}\ e\ :\ C}$$

$$(\Sigma^{\sim}\text{-C}) \quad \frac{\Gamma;*\vdash a:A \quad \Gamma;\Delta\vdash b:[a/x]B \quad \Gamma, x{:}A;\ \Delta', y{:}B\vdash e:C \quad \Gamma;\ *\vdash C\ type}{\Gamma;\ (\Delta,\Delta')\vdash \mathbf{let}\ pair(x,y)=pair(a,b)\ \mathbf{in}\ e\ =\ [a/x,b/y]e\ :\ C}$$

Figure 3: Rules for $\Sigma^{\sim}$-types.

where T_{CN} maps any common noun to a type (we often omit T_{CN} and just write A for $T_{CN}(A)$). (CN is closed under several type constructors including the directed dependent sum types below.)

Example 2.1. *Here is a simple example in syntactic analysis (as in categorial grammar). Consider the following sentence (1):*

(1) Every student works.

The words in the sentence can be given the following types:

(2) $every : \Pi^r X{:}CN.\ S/(X\setminus S)$

(3) $student : CN$

(4) $works : human\setminus S$

where S is the type of sentences and student is a subtype of human (and, hence by contravariance, $human\setminus S$ is a subtype of $student\setminus S$). It is then straightforward to derive that

$$app^r(every, student)\ works\ :\ S$$

In other words, (1) is a sentence.

Directed Dependent Sums Dependent sum types (Σ-types) are split into reverse dependent sums ($\Sigma^{\sim}$) and concatenation dependent sums (Σ°). The rules for $\Sigma^{\sim}$-types are given in Figure 3, while the rules for Σ°-types are symmetric and omitted. We remark that the universe CN is closed under $\Sigma^{\sim}$ and Σ°. For example, directed dependent sum types may be used to analyse modified common nouns when the modifying adjectives are intersective or subsective. To illustrate this with an example, assuming that $B : A\setminus S$, let's use $\Sigma^{\sim}(A,B)$ to abbreviate $\Sigma^{\sim}x{:}A.(x\ B)$ and $\Sigma^{\circ}(A,B)$ to abbreviate $\Sigma^{\circ}x{:}A.(x\ B)$. Now, with $diligent : human\setminus S$, we can use $\Sigma^{\sim}(student, diligent)$ to describe the modified common noun *diligent student*, and $\Sigma^{\circ}(student, diligent)$ to analyse *student who is diligent*.

References

[1] I. Cervesato and F. Pfenning. A linear logical framework. *Inform. and Comput.*, 179(1):19–75, 2002.

[2] S. Chatzikyriakidis and Z. Luo. Natural language reasoning in Coq. *J. of Log, Lang. and Inform.*, 23(4):441–480, 2014.

[3] P. de Groote and S. Maarek. Type-theoretic extensions of abstract categorial grammars. In *Proc. of ESSLLI 2007 Wksh. on New Directions in Type-theoretic Grammars, NDTTG 2007*, pp. 19–30. 2007. `http://let.uvt.nl/general/people/rmuskens/ndttg/ndttg2007.pdf`

[4] P. de Groote, S. Maarek, and R. Yoshinaka. On two extensions of abstract categorial grammars. In *Proc. of LPAR 2007*, v. 4790 of *Lect. Notes in Comput. Sci*, pp. 273–287. Springer, 2007.

[5] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50(1–2):1–102, 1987.

[6] N. Krishnaswami, P. Pradic, and N. Benton. Integrating dependent and linear types. In *Proc. of POPL 2015*, pp. 17–30. ACM, 2015.

[7] J. Lambek. The mathematics of sentence structure. *Amer. Math. Monthly*, 65(3):154–170, 1958.

[8] Z. Luo. Common nouns as types. In *Proc. of LACL 2012*, v. 7351 of *Lect. Notes in Comput. Sci.*, pp. 173–185. Springer, 2012.

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

[10] Z. Luo. Formal semantics in modern type theories: is it model-theoretic, proof-theoretic, or both? In *Proc. of LACL 2014*, v. 8535 of *Lect. Notes in Comput. Sci.*, pp. 177–188. Springer, 2014.

[11] S. Martin and C. Pollard. A dynamic categorial grammar. *Formal Grammar 2014*, 2014.

[12] M. Moortgat. Categorial type logic. In J. van Benthem and A. ter Meulen, eds., *Handbook of Logic and Language*, pp. 3–43. Elsevier, 1997.

[13] G. Morrill. *Categorial Grammar: Logical Syntax, Semantics, and Processing.* Cambridge Univ. Press, 2011.

[14] A. Ranta. *Type-Theoretical Grammar.* Oxford Univ. Press, 1994.

[15] A. Ranta. Syntactic calculus with dependent types. *J. of Log., Lang. and Inform.*, 7(4):413–431, 1998.

[16] M. Vákár. A categorical semantics for linear logical frameworks. In *Proc. of FoSSaCS 2015*, v. 9034 of *Lect. Notes in Comput. Sci.*, pp. 102-116. Springer, 2015.

[17] D. Walker. Substructural type systems. In B. Pierce, ed., *Advanced Topics in Types and Programming Languages*, pp. 3–43. MIT Press, 2005.

Introducing a Type-theoretical Approach to Universal Grammar

Erkki Luuk

Institute of Computer Science, University of Tartu, Estonia
erkkil@gmail.com

Abstract

The idea of Universal Grammar (UG) as the hypothetical linguistic structure shared by all human languages harkens back at least to the 13th century. The best known modern elaborations of the idea are due to Chomsky. Following a devastating critique from theoretical, typological and field linguistics, these elaborations, the idea of UG itself and the more general idea of language universals stand untenable and are largely abandoned. The presentation shows how to tackle the hypothetical structure of UG in a framework very different from the Chomskyan ones, using dependent and polymorphic type theory. The expected outcome of this work in progress is a versatile logic for expressing natural language morphosyntax and compositional semantics.

1 Universal Grammar

Type theory, in its modern form (i.e. as dependent and/or polymorphic type theory), is the most expressive logical system (as compared to the nonlogical ones such as set and category theory). As [6, 2, 3, 1] have shown, complex type theories outshine simpler ones in accounting for natural language (NL) phenomena like anaphora, selectional restrictions, etc. Second, as the notion of type is inherently semantic (type := class of semantic values), it is by definition suited for analyzing universal phenomena in NL, the semantics of which is largely universal (as witnessed by the possibility of translation from any human language to another).

The talk introduces a notationally simple system (L) matching NL morphosyntax as closely as possible. The purpose of L is to express the universal core of NL morphosyntax in a formally and linguistically precise, yet parsimonious way. L is furnished with elementary and complex morphosyntactic types, some of which are universal, others nonuniversal or possibly universal. The system is based on dependent and polymorphic type theory, with Martin-Löf's type theory (MLTT; [5]) as the main type-theoretical reference point. L is a relational logic in prefix notation, with formulas like RUN(THE(Y(man))), (Y(LOVE))(mary,john), etc. (0^{th} order relations (1^{st} order arguments) in small letters, all other relations in capitals; Y a tense/person/number (etc.) marker). Elementary relation symbols represent constant types inhabited by language-particular (proof) objects (e.g. the type 'man' has objects $man, homme$, etc.). Both elementary and complex formulas of L are morphosyntactic types that have semantics (by the definition of type and by their alignment with morphosyntactic categories that have semantics (also by definition)). By a wff of L we mean one that is wf both syntactically and semantically, i.e. wf and well-typed. In L's universe $\mathcal{U}$, all formulas are morphosyntactic types and all types are morphosyntactic formulas (by Curry-Howard isomorphism).

L is capable of analyzing anaphoric devices (pronouns, agreement, complementizers), linguistic quantifiers (straightforwardly, since quantifiers are higher-order relations), selectional restrictions (in par with [2, 1], but more directly, with formulas like $\text{READ}_{\text{I,S}}(x, y)$, where capital subscript letters impose type restrictions on relations argument(s) ($\text{READ}_{\text{I,S}}(x, y)$ means that x is restricted to type I and y to type S)), etc. The centerpiece of L are typing rules,

some simple examples of which are o(N, PN, PRO, X/*, GER) : X; $this, that, those, these...$: DEM; $a, the, other...$: DET; o(DET, DEM) : D (o($\boldsymbol{x}$) := all terms of type(s) $\boldsymbol{x}$ ($\boldsymbol{x}$ a sequence of types)). Largely because of L's metalanguage operators |, *, [], /, etc., only 30-50 rules are required for a (possibly) complete specification of UG.

2 Type theory

Two main kinds of polymorphism in NL are term underspecification ("data type polymorphism" in computer science) and argument ambiguity (e.g., "subtype polymorphism" in computer science). The latter is handled with function o (Sec. 1), since a judgement $o(A_1, ..., A_n) : C$ is a polymorphic binary relation of type $(X \in A_1, ..., A_n, C)$. MLTT handles polymorphism with universes, so in MLTT one would fix a universe $U = A_1, ..., A_n$ and the relation's type would be written $(\sum X \in U)C$ or $\sum(U, C)$.

We give an example of casting L's formulas to MLTT. Posit universes **Rel** and **Arg** (for morphosyntactic relation and argument, resp.). Write an L formula $A(B)$ ($A : \mathbf{Rel}, B : \mathbf{Arg}$). Since arguments depend on relations, we need dependent types, i.e. a modern type theory such as MLTT. Per MLTT's notation, $A(B)$ gets the preliminary form (A, B). The main dependent type constructors in MLTT are $\sum$ and $\prod$. With (A, B), $\prod$ would reverse the natural order by taking **Rel** for the domain and **Arg** for codomain, so we will use $\sum$ instead. Thus the formula map from L to MLTT is $A(B) \mapsto \sum(A, B)$.

Term underspecification is resolved by /-types (cf. [4]). An underspecified term b will be typed with a /-separated string of symbols of all types over which b's type is polymorphic. A /-type is defined wrt. at least two other types $C_1, \ldots, C_n$ between which no subtyping relation holds.

The description of UG (and more generally, NL grammar) with dependent and polymorphic types is very different from the traditional Chomskyan approximation of UG with rewrite rules and/or syntax trees. The main differences from existing type-theoretical approaches (e.g. [6, 2, 3, 1]) are a focus on UG, an integrated treatment of morphosyntax and sentential/phrasal semantics, and an employment of polymorphic and dependent types.

Acknowledgement The author thanks Erik Palmgren and Roussanka Loukanova. This work was supported by IUT20-56 "Computational models for Estonian", the Swedish Institute, and the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS.

References

[1] N. Asher. Selectional restrictions, types and categories. *J. of Appl. Log.*, 12(1):75–87, 2014.

[2] Z. Luo. Type-theoretical semantics with coercive subtyping. In N. Li and D. Lutz, eds., *Proc. of 20th Semantics and Linguistic Theory Conf., SALT 2010*, pp. 38–56. 2010.

[3] Z. Luo. Formal semantics in modern type theories: is it model-theoretic, proof-theoretic, or both? In N. Asher and S. Soloviev, eds., *Proc. of 8th Int. Conf. on Logical Aspects of Computational Linguistics, LACL 2014*, v. 8535 of *Lect. Notes in Comput. Sci.*, pp. 177–188. Springer, 2014.

[4] E. Luuk. Nouns, verbs and flexibles: implications for typologies of word classes. *Lang. Sci.*, 32(3):349–365, 2010.

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

[6] A. Ranta. *Type-Theoretical Grammar.* Clarendon Press, 1994.

The Encode-Decode Method, Relationally

James McKinna[1] and Fredrik Nordvall Forsberg[2]

[1] School of Informatics, University of Edinburgh
[2] Dept. of Comput. and Inform. Sci., University of Strathclyde
james.mckinna@ed.ac.uk, fredrik.nordvall-forsberg@strath.ac.uk

Abstract

In Homotopy Type Theory (HoTT), the 'encode-decode' method makes heavy use of explicit recursion over higher inductive types to construct, and prove properties of, homotopy equivalences. We argue for the classical separation between specification and implementation, and hence for using relations to track the graphs of encode/decode functions. Our aim is to isolate the technicalities of their definitions, arising from higher path constructors, from their properties. We describe our technique in the calculation of $\pi_1(\mathbb{S}^1)$, and comment on its applicability in the current Agda implementation of HoTT.

Introduction Burstall's seminal analysis [2] of how to prove properties of fold (defined by structural *recursion* on lists) proceeds by first deriving a proof principle for fold by structural *induction*, and then applying it. This avoids relying on the *definition* of fold to achieve coincidences in proof by list induction, by encapsulating the recursive call structure once and for all. This achieves separation of concerns between a *concrete* implementation and its *abstract* specification in terms of the induction principle for its *graph*. This insight has been rediscovered, applied, and engineered many times over; in type theory most recently in the `Function` [1] and `Program` [8] extensions to Coq, and in the design and implementation of *views* in Epigram [6].

The *encode-decode* method, pioneered by Licata and Shulman in their proof of $\pi_1(\mathbb{S}^1) \simeq \mathbb{Z}$ [4], heavily exploits definition by structural recursion and proof by induction on the newly-introduced Higher Inductive Types (HITs) of Homotopy Type Theory (HoTT [9]), where inductive types introduce not only new *term* constructors, but also those of (higher) *paths*. Below we explore Burstall's technique in this context, and its implementation in systems such as Agda.

Graphs of (recursively-defined) functions A function $f : \Pi_{x:A} B(x)$ gives rise to a *graph* relation $F : \Pi_{x:A}\Pi_{y:B(x)}\mathsf{Type}$ trivially via $Fxy \equiv y = f(x)$. But if f is defined *recursively*, then F may be given *inductively*, with base cases of the definition of f corresponding to axioms in that of F, and step cases to inference rules, with recursive calls of f tracked by inductive premises involving F. Such an F then yields an induction principle for f, and proofs that F is *sound*, $\mathsf{snd}_f(F) : \Pi_{x:A}\Pi_{y:Bx} Fxy \to y = f(x)$, and *complete*, $\mathsf{cmp}_f(F) : \Pi_{x:A} Fx(f(x))$, for the graph relation. Turning this around, *any* choice of (inductive) family G satisfying $\mathsf{snd}_f(G)$ and $\mathsf{cmp}_f(G)$ constitutes an adequate representation of the graph of f, and may be used in proofs about f, by virtue of establishing that F and G are *extensionally equivalent* ($\Leftrightarrow$) as relations.

The encode-decode equivalence, revisited The circle $\mathbb{S}^1$ is given as a HIT with 0-constructor $\mathsf{base} : \mathbb{S}^1$ and 1-constructor $\mathsf{loop} : \mathsf{base} = \mathsf{base}$. Its *covering space* $C(x)$ is defined by higher recursion on $x : \mathbb{S}^1$, by $C(\mathsf{base}) \equiv \mathbb{Z}$ and a path $\mathbb{Z}{=}\mathbb{Z}$ obtained via the Univalence Axiom from the automorphism of $\mathbb{Z}$ induced by $\mathsf{succ} : \mathbb{Z} \to \mathbb{Z}$. Then the path space $P(x) \equiv \mathsf{base} = x$ over $x : \mathbb{S}^1$ is shown homotopy equivalent to $C(x)$ via functions $\mathsf{encode} : \Pi_{x:\mathbb{S}^1}\Pi_{p:P(x)} C(x)$ (given by the action $p^\star(0) \equiv \mathsf{transport}_{C(\_)}p(0)$ of paths p on $0 : \mathbb{Z}$) and $\mathsf{decode} : \Pi_{x:\mathbb{S}^1}\Pi_{c:C(x)} P(x)$, defined by higher induction on $\mathbb{S}^1$: the function $\mathsf{decode}_{\mathsf{base}}$ maps $z : \mathbb{Z}$ to the iterated path loop^z, and one has to show this definition respects the action $\mathsf{loop}^\star$ of loop.

Our approach requires the choice of two relations, $\mathsf{Encode}_x\ p\ c$ and $\mathsf{Decode}_x\ p\ c$, which we show sound and complete for encode_x and decode_x, together with a proof of the homotopy equivalence, which amounts to proving: $\Pi_{x:\mathbb{S}^1}\Pi_{p:P(x)}\Pi_{c:C(x)}\mathsf{Encode}_x\ p\ c \Leftrightarrow \mathsf{Decode}_x\ c\ p$ (†). For $\mathsf{Encode}_x\ p\ c$ we take simply the graph of encode_x, while for $\mathsf{Decode}_{\mathsf{base}}\ z\ p$ we take the inductively defined graph of $z \mapsto \mathsf{loop}^z$. The main subtlety lies in proving $\mathsf{cmp}_{\mathsf{decode}_x}(\mathsf{Decode}_x)$. As above, one must show that $\mathsf{Decode}_{\mathsf{base}}$ respects the $\mathsf{loop}^\star$ action in an appropriate sense. We then have that decode_x and encode_x are inverse: by $\mathsf{cmp}_{\mathsf{encode}_x}(\mathsf{Encode}_x)$, we have $\mathsf{Encode}_x\ p\ (\mathsf{encode}_x(p))$, hence $\mathsf{Decode}_x\ (\mathsf{encode}_x(p))\ p$ by (†), and finally $\mathsf{decode}_x(\mathsf{encode}_x(p)) = p$ by $\mathsf{snd}_{\mathsf{decode}_x}(\mathsf{Decode}_x)$. The other direction is similar.

Now, the argument may be further streamlined: since we *choose* relations Encode_x and Decode_x, subject to the equivalence (†), we might as well take $\mathsf{Encode}_x\ p\ c \equiv \mathsf{Decode}_x\ c\ p$, and thus must show $\mathsf{Decode}_x\ c\ p$ sound and complete for encode_x, or else $\mathsf{Decode}_x\ c\ p \equiv \mathsf{Encode}_x\ p\ c$, and show that $\mathsf{Encode}_{\mathsf{base}}$ is closed under $\mathsf{loop}^\star$, and Encode_x sound and complete for decode_x.

Work in progress! Towards validating the above approach, and making comparisons with Licata-Shulman, we have made some progress on an Agda formalisation [7], subject to some wrinkles: firstly, inductive families such as those considered above appear *not* well-tolerated by Agda, and in particular, its pattern-matching algorithm; instead, one must give *equivalent* formulations, where the conclusions are explicitly described by equational premises [5]. Secondly, the proofs of soundness and completeness are made far heavier by the explicit appeal to higher induction on $\mathbb{S}^1$, in particular when showing closure of $\mathsf{Encode}_{\mathsf{base}}$, $\mathsf{Decode}_{\mathsf{base}}$ under $\mathsf{loop}^\star$.

Conclusions and future work We have only had time (and space!) to explore one of the simplest instances of the encode-decode method. We hope to extend our approach to examples such as Brunerie's Flattening Lemma [9, Lemmas 8.1.12, 6.12.2], the Freudenthal suspension theorem [*ibid.*, Theorem 8.6.4], or Cavallo's analysis of the Mayer-Vietoris sequence [3], and to consider how systems such as Agda might better support the methods described here.

Acknowledgments We are grateful for feedback from the TYPES referees and the recent Strathclyde HoTT workshop, as well as EPSRC support (EP/K020218/1 and EP/K023837/1).

References

[1] G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In *Proc. of FLOPS 2006*, v. 3945 of *Lect. Notes in Comput. Sci.*, pp. 114–129. Springer, 2006.

[2] R. M. Burstall. Proving Properties of Programs by Structural Induction. *Comput. J.*, 12(1):41–48, 1969.

[3] E. Cavallo. Exactness of the Mayer-Vietoris sequence in homotopy type theory. Draft, 2015. `http://www.contrib.andrew.cmu.edu/~ecavallo/works/mayer-vietoris.pdf`

[4] D. Licata and M. Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Proc. of LICS '13*, pp. 223–232. IEEE CS, 2013.

[5] C. McBride. Inverting inductively defined relations in LEGO. In *Selectd Papers from TYPES '96*, v. 1512 of *Lect. Notes in Comput. Sci.*, pp. 236–253. Springer, 1998.

[6] C. McBride and J. McKinna. The view from the left. *J. of Funct. Program.*, 14(1):69–111, 2004.

[7] J. McKinna and F. Nordvall Forsberg. `http://homepages.inf.ed.ac.uk/jmckinna/LoopSpaceCircleREL.agda`

[8] M. Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université Paris 11, 2008.

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

Toward Dependent Choice: A Classical Sequent Calculus with Dependent Types

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

[1] Laboratoire PPS, INRIA and Université Paris 7, France
[2] IMERL, Universidad de la República, Montevideo, Uruguay
hugo.herbelin@inria.fr, emiquey@pps.univ-paris-diderot.fr

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

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

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

We present here a continuation of Herbelin's works [5], who proposed a way of scaling up Martin-Löf proof to classical logic. The first idea is to restrict the dependent sum type to a fragment of our system we call *N-elimination-free*, making it computationally compatible with classical logic. The second idea is to represent a countable universal quantification as an infinite conjunction. This allows to internalize into a formal system (called dPA^ω) the realizability approach [2, 4] as a direct proof-as-programs interpretation.

Informally, let us imagine that given $H : \forall x^A \exists y^B P(x,y)$, we have the ability of creating an infinite term $H_\infty = (H0, H1, \ldots, Hn, \ldots)$ and select its n^{th}-element with some function `nth`. Then one might wish that

$$\lambda H.(\lambda n.\,\mathtt{wit}(\mathtt{nth}\ n\ H_\infty), \lambda n.\,\mathtt{prf}(\mathtt{nth}\ n\ H_\infty))$$

could stand for a proof for $AC_{\mathbb{N}}$. However, even if we were effectively able to build such a term, H_∞ might contain some classical proof. Therefore two copies of H_n might end up being different according to their context in which they are executed, and then return two different witnesses. This problem could be fixed by using a shared version of H_∞, say

$$\lambda H.\,\mathtt{let}\, a = H_\infty\, \mathtt{in}\ (\lambda n.\,\mathtt{wit}(\mathtt{nth}\ n\ a), \lambda n.\,\mathtt{prf}(\mathtt{nth}\ n\ a)\,.$$

It only remains to formalize the intuition of H_∞. We do this by a stream $\mathtt{cofix}^0_{fn}(Hn, f(S(n)))$ iterated on f with parameter n, starting with 0 :

$$AC_{\mathbb{N}} := \lambda H.\,\mathtt{let}\, a = \mathtt{cofix}^0_{fn}\ (Hn, f(S(n))\,\mathtt{in}(\lambda n.\,\mathtt{wit}(\mathtt{nth}\ n\ a), \lambda n.\,\mathtt{prf}(\mathtt{nth}\ n\ a)\,.$$

Whereas the stream is, at level of formulæ, an inhabitant of a coinductive defined infinite conjunction $\nu^0_{Xn}(\exists P(0,y) \wedge X(n+1))$, we cannot afford to pre-evaluate each of its components, and then have to use a *lazy* call-by-value evaluation discipline. However, it still might be responsible for some non-terminating reductions. Our approach to prove a normalization property would be to interpret it in HA^ω through a negative translation. However, the sharing forces us to have a state-passing-style translation, whose small-step behaviour is quite far from the sharing strategy we have in natural deduction.

In a recent paper, Ariola *et al.* presented a way to construct a CPS-translation for a call-by-need version of the $\bar{\lambda}\mu\tilde{\mu}$-calculus [1], which allows some sharing facilities. Yet, this translation

does not enjoy any typing property, and then does not give us a way of proving normalization. Moreover, the $\bar{\lambda}\mu\tilde{\mu}$-calculus is typed with sequent calculus [3], which does not allow to manipulate dependent types immediately.

We propose to deal with both problems while proving the normalization of our system in two steps. First, we translate our calculus to an adequate version of the $\bar{\lambda}\mu\tilde{\mu}$-calculus that allows to manipulate dependent types on the N-elimination-free fragment. Then we will try to adapt the CPS-translation for call-by-need to our case, while adding it a type.

This work is currently in progress. For now, we managed to tackle the first problem, that is to construct a sequent calculus version of the initial language dPA^ω, During this talk, we intend to focus on this point, which turns out to be tricky, mainly because of the desynchronization of the dependency at the level of type in call-by-value. Let us look at the β-rule to get an insight of what happens. If we define the $\to_L$ and $\to_R$ rule as expected (where $A^{\perp\!\!\!\perp}$ is the type of a refutation of A):

$$\frac{\Gamma, a : A \vdash p : B}{\Gamma \vdash \lambda a.p : [a : A] \to B}\ \to_L \qquad \frac{\Gamma \vdash q : A \quad \Gamma \vdash e : B[q/a]^{\perp\!\!\!\perp} \quad q \notin \mathrm{Nef} \to a \notin A}{\Gamma \vdash q \cdot e : ([a : A] \to B)^{\perp\!\!\!\perp}}\ \to_R$$

and consider such a proof $\lambda a.p : [a : A] \to B$ and a context $q \cdot e : [a : A] \to B$, it reduces as follows:

$$\langle \lambda a.p \mid q \cdot e \rangle \rightsquigarrow \langle q \mid \tilde{\mu} a.\langle p \mid e \rangle \rangle$$

On the right side, we see that p, whose type is $B[a]$, is now cut with e of type $B[q]$. The idea is that in the full command a has been linked to q at a previous level of the typing judgement. We fixed this problem by making explicit a dependency list in the typing rules, which allows this typing derivation:

$$\dfrac{\Gamma \vdash q : A \qquad \dfrac{\dfrac{\Gamma, a : A \vdash p : B[a] \quad \Gamma, a : A \vdash e : B[q]; \{a|q\}}{\langle p \mid e \rangle : \Gamma, a : A; \{a|q\}}\ \text{CUT}}{\Gamma \vdash \tilde{\mu} a.\langle p \mid e \rangle : A^{\perp\!\!\!\perp}; \{.|q\}}\ \tilde{\mu}}{\langle q \mid \tilde{\mu} a.\langle p \mid e \rangle \rangle : \Gamma; \{\cdot|\cdot\}}\ \text{CUT}$$

By using this dependency list, we managed to fully translate the dPA^ω of [5] into a sequent calculus framework, that is a $\bar{\lambda}\mu\tilde{\mu}$-calculus with treatment of induction, cofix and equality. The translation is fully correct with respect to types.

The resulting calculus is given with a head-reduction, following a call-by-need evaluation strategy, and makes explicit the shared environment. This makes it a lot more closer to a small-step abstract machine than the original calculus, and it is our hope that as in [1] this would make the construction of a correct CPS-translation easier.

References

[1] Z. M. Ariola, P. Downen, H. Herbelin, K. Nakata, and A. Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In *Proc. of FLOPS 2012*, v. 7294 of *Lect. Notes in Comput. Sci.*, pp. 32–46. Springer, 2012.

[2] S. Berardi, M. Bezem, and T. Coquand. On the computational content of the axiom of choice. *J. of Symb. Log.*, 63(2):600–622, 1998.

[3] P.-L. Curien and H. Herbelin. The duality of computation. In *Proc. of ICFP 2000*, pp. 233–243. ACM, 2000.

[4] M. H. Escardó and P. Oliva. Bar recursion and products of selection functions. arXiv:1407.7046, 2014.

[5] H. Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *Proc. of LICS '12*, pp. 365–374. IEEE CS, 2012.

Π-Ware: An Embedded Hardware Description Language using Dependent Types

João Paulo Pizani Flor and Wouter Swierstra

Dept. of Inform. and Comput. Sci., Universiteit Utrecht, The Netherlands
{j.p.pizaniflor|w.s.swierstra}@uu.nl

Computer hardware has experienced a steady exponential increase in complexity in the last decades. There is an increased demand for application-specific integrated circuits that avoid the proverbial *von Neumann bottleneck* [1], and ever more algorithms enjoy hardware acceleration (such as 3D/2D renderers, video/audio codecs, cryptographic primitives and network protocols).

This demand puts pressure on the industry to make hardware design quicker and more efficient. At the same time, hardware design also requires very strong correctness guarantees. These strong correctness requirements are commonly met using (exhaustive) testing and model checking, making the design-verify-fix loop slower and more costly than in the software world.

There is a long tradition [4, 6] of modeling hardware using functional programming to pursue higher productivity and stronger correctness assurance. A particular trend is to use functional programming languages as *hosts* for Embedded Domain-Specific Languages (EDSLs) aimed at hardware description, of which Lava [2] is a popular example. Some of the limitations of these hardware EDSLs are due to the host's type system, which lacks the power to express some desirable properties of circuit models.

We believe that the advantages brought to hardware design by FP-inspired techniques can be even greater if we use dependent types. We define a Hardware Description Language (HDL) called Π-Ware, which is *embedded* in the dependently-typed general-purpose programming language Agda. Circuits described in Π-Ware can be simulated, transformed in several ways, proven correct, and synthesized (work in progress).

The existence of several different semantics for circuit models is due to Π-Ware's *deep embedding*: There is an inductive datatype of circuits (called C), and semantics (simulation, synthesis, gate count, etc.) are just functions with C as domain. More specifically, the circuit datatype is *indexed* by two natural numbers representing, respectively, the *sizes* of the circuit's input and output. The arithmetic fine tuning of these indices, along with other features of dependent types, allow us to "ban" certain classes of design mistakes *by construction*, for example:

- Our circuit constructors guarantee that all circuits are *well-sized*, i.e., there are never "floating" or "dangling" wires.
- Connections between circuits are guaranteed to never cause *short-circuits* (two or more sources connected to the same load).

In addition to this *structural correctness by construction*, we can also *interactively* prove properties of circuits, which provides some advantages over the fully-automated verification approach used widely in industry. First of all, we can inductively prove properties over whole *circuit families*. Furthermore, proofs written in Agda are *modular*, in contrast to fully-automated verification. This means that, when proving laws concerning a certain circuit (family), we *reuse* previously proven facts about its constituent subcircuits.

In particular, we can write proofs of *functional correctness*. Our simulation semantics (even for sequential circuits) are *executable* (in contrast to other EDSLs [3] with a *relational* semantics). This means that the simulation semantics simply converts each circuit into a function,

which can then be applied to input vectors, producing output vectors. We can then formulate the statement of whether a circuit (family) *implements* the behaviour of a given *specification function*, and proofs of correctness can be written by induction on circuit inputs.

As a first step in exploring the possibilities that our approach offers, we conducted a case study, aiming to formalize a class of circuits known as *parallel-prefix sums* [5] in Π-Ware. Parallel-prefix circuit combinators have been defined *in terms of* Π-*Ware primitives*, and we are proving several properties of this class of circuits which had previously only been postulated or proven on paper.

The case study has led us to establish suitable *equivalence relations* between circuits. We have defined a relation of *equality up to simulation*, which identifies any two circuits with the same simulation behaviour (taking equal inputs to equal outputs). Reasoning about this equivalence relation, without postulating function extensionality, presents several challenges, for which we believe to have found satisfactory solutions. Finally, we show how to define *algebraic* laws involving circuit constructors and combinators. These may be used to construct *provably safe* circuit transformations.

All in all, we claim there are several benefits to be gained from using dependent types to describe hardware circuits. These techniques are more widely applicable to other domains and, therefore, we believe them to be of interest to the wider TYPES community.

References

[1] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. of ACM*, 21(8):613–641, 1978.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. *SIGPLAN Not.*, 34(1):174–184, 1998.

[3] T. Braibant. Coquet: a Coq library for verifying hardware. In J.-P. Jouannaud and Z. Shao, eds., *Proc. of CPP 2011*, v. 7086 in *Lect. Notes in Comput. Sci.*, pp. 330–345. Springer, 2011.

[4] P. Gammie. Synchronous digital circuits as functional programs. *ACM Comput. Surv.*, 46(2):21, 2013.

[5] R. Hinze. An algebra of scans. In D. Kozen, ed., *Proc. of MPC 2004*, v. 3125 in *Lect. Notes in Comput. Sci.*, pp. 186–210. Springer, 2004.

[6] M. Sheeran. Hardware design and functional programming: a perfect match. *J. of UCS*, 11(7):1135–1158, 2005.

Well-Founded Sized Types in the Calculus of (Co)Inductive Constructions *

Jorge Luis Sacchini

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

Type-based termination is a mechanism for ensuring termination and productivity of (co)recursive definitions [4]. Its main feature is the use of sized types (i.e. types annotated with size information) to track the size of arguments in (co)recursive calls. Termination of recursive function (and productivity of corecursive functions) is ensured by restricting recursive calls to smaller arguments (as evidenced by their types). Type-based approaches to termination and productivity have several advantages over the syntactic-based approaches currently implemented in Coq and Agda [5, 1, 2]. In particular, they are more expressive and easier to understand.

In previous work [6, 5], we studied the metatheory of extensions of the Calculus of (Co)Inductive Constructions (CIC) with a simple notion of sized types. While these systems are more expressive than the guard predicates of Coq, there have two main shortcomings in the case of coinductive types. First, corecursive definitions that use tail are not expressible, e.g.: $\mathsf{corec\ zeroes} := \mathsf{cons}(0, \mathsf{cons}(0, \mathsf{tail\ zeroes}))$.

Second, Subject Reduction (SR) does not hold (a well-known problem in Coq that was observed by Gimnez [3]). While in practice, the lack of SR does not seem to be a major issue, it is theoretically unsatisfying.

In this work, we consider an extension of CIC with a notion of well-founded sized types, inspired by $\mathsf{F}^{\mathsf{cop}}_{\omega}$ [2], that solves both issues. As in previous work, (co)inductive types are annotated with size expressions taken from the size algebra given by $s ::= \imath \mid s + 1 \mid \infty$, where $\imath$ is a size variable. In this approach, the (simplified) typing rule for streams looks like:

$$\frac{f : [\jmath < \imath]\, \mathsf{stream}^{\jmath}\, A \vdash t : \mathsf{stream}^{\imath}\, A \qquad \imath \text{ fresh}}{\vdash (\mathsf{corec}\, f := t) : \forall \imath . \mathsf{stream}^{\imath}\, A}$$

The type $\mathsf{stream}^{s}\, A$ is the type of streams (whose elements have type A) where at least s elements can be produced. The definition above ensures that t can only make (co)recursive calls on smaller (i.e. already produced) streams. Since $\jmath < \imath$ we ensure that at least one more element is produced by t. We require that each corecursive call to f be explicitly applied to a size s satisfying the constraint $s < \imath$.

The type of the stream constructor, cons, is (informally) the following:

$$\mathsf{cons} : (\forall \jmath < s . A \times \mathsf{stream}^{\jmath}\, A) \to \mathsf{stream}^{s}\, A$$

To construct an element of type $\mathsf{stream}^{s}\, A$ we need to provide the head of type A and the tail of type $\mathsf{stream}^{\jmath}$ for an arbitrary $\jmath < s$. To destruct cons above, we must apply it to a size r satisfying $r < s$.

*This publication was made possible by a JSREP grant (JSREP 4-004-1-001) from the Qatar National Research Fund (a member of The Qatar Foundation). The statements made herein are solely the responsibility of the author.

Let us see how well-founded sized types overcome the first limitation of our previous approach mentioned above. Consider the following definition of the stream of Fibonacci numbers:

$$\mathsf{FIB} \equiv \mathsf{corec}\,\mathsf{fib} := \mathsf{cons}(\jmath.\,0, \mathsf{cons}(\kappa.\,1, \mathsf{sum}[\kappa]\,\mathsf{fib}[\kappa]\,(\mathsf{tail}[\kappa]\,\mathsf{fib}[\jmath])))$$

where, given $t : \mathsf{stream}^s\,A$, $\mathsf{tail}[r](t) : \mathsf{stream}^r\,A$ for any $r < s$, and $\mathsf{sum} : \forall\imath.\mathsf{stream}^\imath\,\mathsf{nat} \to \mathsf{stream}^\imath\,\mathsf{nat} \to \mathsf{stream}^\imath\,\mathsf{nat}$ computes the point-wise addition of two streams of natural numbers.

This definition is well typed. The size variables introduced ($\jmath$ and κ) satisfy $\kappa < \jmath$ and $\jmath < \imath$. The recursive call $\mathsf{fib}[\jmath]$ has type $\mathsf{stream}^\jmath\,\mathsf{nat}$, and $\mathsf{tail}[\kappa]\,\mathsf{fib}[\jmath]$ has type $\mathsf{stream}^\kappa\,\mathsf{nat}$.

Let us consider the second issue mentioned above: lack of SR. In CIC (and Coq), unfolding of (co)recursive definitions is restricted in order to have a strongly normalizing evaluation. In the case of Coq, unfolding of corecursive definitions is only allowed within case analysis (which leads to the loss of SR).

With well-founded sized types, we use the size annotations to restrict unfolding. Corecursive functions have types of the form $\forall\imath.\mathsf{stream}^\imath\,A$ (in general, $\forall\imath.\Pi\Delta.\mathsf{stream}^\imath\,A$). Like corecursive calls, we require that corecursive functions be applied to a size annotation (size application is denoted $t[s]$). We restrict unfolding to the case where the size annotation is ∞:

$$(\mathsf{corec}\,f := t)[s] \to t[f/\mathsf{corec}\,f := t] \quad \text{if } s = \infty$$

This reduction strategy satisfies SR (the proof is standard) [7]. Furthermore, we have proved that this reduction relation is strongly normalizing. For instance, it prevents infinite unfolding of $\mathsf{tail}\,\mathsf{FIB}$:

$$\begin{aligned}\mathsf{tail}[\infty]\,\mathsf{FIB}[\infty] &\to \mathsf{tail}[\infty]\ (\mathsf{cons}(\jmath.\,0, \mathsf{cons}(\kappa.\,1, \mathsf{sum}[\kappa]\,\mathsf{fib}[\kappa]\,(\mathsf{tail}[\kappa]\,\mathsf{fib}[\jmath]))))\\ &\to \mathsf{cons}(\kappa.\,1, \mathsf{sum}[\kappa]\,\mathsf{fib}[\kappa]\,(\mathsf{tail}[\kappa]\,\mathsf{fib}[\infty]))\\ &\to \mathsf{cons}(\kappa.\,1, \mathsf{sum}[\kappa]\,\mathsf{fib}[\kappa]\,(\mathsf{cons}(\kappa'.\,1, \mathsf{sum}[\kappa']\ \mathsf{fib}[\kappa'](\mathsf{tail}[\kappa']\,\mathsf{fib}[\kappa]))))\end{aligned}$$

Unfolding cannot proceed until the stream is further observed (e.g. by applying tail again).

This work is still in progress; the current version is available at [7]. We have proved most of the metatheory, including strong normalization. We have also proved that size-inference is decidable, i.e. size annotation can be inferred from minimal user-provided annotations, and we implemented a small prototype (available at [7]). Our prototype already performs size inference; however, the algorithm used is very inefficient. We are currently investigating more efficient alternatives.

References

[1] A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[2] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In *Proc. of ICFP '13*, pp. 185–196. ACM, 2013.

[3] E. Giménez. *A Calculus of Infinite Constructions and Its Application to the Verification of Communicating Systems*. PhD thesis, École Normale Supérieure de Lyon, 1996.

[4] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proc. of POPL '96*, pp. 410–423. ACM, 1996.

[5] J. L. Sacchini. *On Type-Based Termination and Dependent Pattern Matching in the Calculus of Inductive Constructions*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2011.

[6] J. L. Sacchini. Type-based productivity of stream definitions in the calculus of constructions. In *Proc. of LICS '13*, pp. 233–242. IEEE CS, 2013.

[7] J. L. Sacchini. On well-founded sized types in the calculus of (co)inductive constructions. Draft, 2015. `http://www.qatar.cmu.edu/~sacchini/well-founded/`

On Relating Indexed W-Types with Ordinary Ones

Christian Sattler

School of Mathematics, University of Leeds, United Kingdom
c.sattler@leeds.ac.uk

Introduction Indexed W-types model inductive families, just as ordinary W-types model inductive types. In the context of extensional type theory, indexed W-types were shown constructible from ordinary ones by Gambino and Hyland in a past TYPES post-proceedings [3] using equivalent categorical terms: they construct initial algebras for dependent polynomial functors from non-dependent ones in a locally cartesian closed category. In intensional type theory with function extensionality[1], an analogous result should hold when considering the corresponding homotopified notion [1] of (indexed) W-types.[2]

Though tedious, this is provable using ad-hoc term-level manipulations following essentially the idea from the extensional case. Instead, we want to highlight a conceptually clean alternative. By illuminating a deeper categorical nature of the extensional construction [3], we make it amendable to higher categorical generalization in terms of locally cartesian closed quasi-categories.

Recent work, partially in progress, by Szumiło [5] and Kapulkin exhibits the syntax of intensional type theory with function extensionality as a locally cartesian closed quasi-category. After verifying that quasi-categorical notions like initial objects in algebra quasi-categories agree with their counterparts defined in the internal language of type theory, this should prove the desired result.

It is noteworthy this approach lets us leave the realm of type-theoretic syntax by working in the semantic domain of quasi-categories. It does not seem possible to formalize internally the infinite tower of coherence e.g. of the notion of algebra morphisms with their compositional structure. Nor is it needed: as realized early on in homotopy type theory, contractibility is internally expressible, letting us define notions like homotopy initial algebra by referencing only the first few levels [1]. Hence, only finitely many levels of coherence will be needed at any point.

However, several steps in a type-theoretic proof would each require explicating an additional layer of coherence to start with, making a manual translation to an internal proof rather infeasable and unreadable (and of little conceptual value).[3] This is due to a deficiency of current syntax for homotopy type theory to adequately capture higher-dimensional categorical coherence in a way that is comparable to how the identity type captures higher-dimensional groupoidal coherence.

Work in progress.

Sketch: the extensional case Since our main contribution is introducing abstraction to a previously concrete argument [3], the use of categorical language in the following sketch is unavoidable. However, we only want to give some intuition for the underlying ideas. We have retricted us to the extensional setting, and the reader is invited to take from it whatever they want.

[1] That is: homotopy type theory without requiring a universe.

[2] Computation and uniqueness laws are formulated using coherent propositional equality, making them homotopy initial algebras.

[3] We conjecture that an effective proof term is generatable from the quasi-categorical proof.

Let $\mathcal{C}$ be a locally cartesian closed. Given $f : B \to A$ and $s : B \to I$ and $t : A \to I$, the associated dependent polynomial endofunctor $[\![f_{s,t}]\!] : \mathcal{C}/I \to \mathcal{C}/I$ is composed of base change along s, dependent product along f, and dependent sum along t:

$$\mathcal{C}/I \xrightarrow{\Delta_s} \mathcal{C}/B \xrightarrow{\Pi_f} \mathcal{C}/A \xrightarrow{\Sigma_t} \mathcal{C}/I$$

The non-dependent version arises as the special case $I = 1$, in which case we will omit s and t.

The basic idea of [3] for constructing the initial algebra $\mu F = \mu\,[\![f_{s,t}]\!]$ is to carve it out of $\mu\,[\![f]\!]$, which we regard as the type of well-founded trees possibly ill-typed with respect to the I-indexing. This is done by taking the equalizer of the diagram $\mu\,[\![f]\!] \rightrightarrows \mu\,[\![I \times f]\!]$ where the two maps copy to each node the index expected from "below" and given by "above", respectively; taking the equalizer corresponds to type-checking the indexing information. The construction of these maps is very much hands-on via explicit recursive definitions, with a lack of symmetry between the two maps and an ad-hoc choice for the index value at the root of the "below" map. This suggests that conceptually we ought to be working in a different slice.

Abbreviate $F = [\![f_{s,t}]\!]$ and write $[\![f]\!] \cong \Sigma_I F \Delta_I$ and $[\![I \times f]\!] \cong \Sigma_I \Delta_I \Sigma_I F \Delta_I$ using 2-functoriality and Beck-Chevalley conditions. Abbreviating $T = \Delta_I \Sigma_I$, the rolling rule [2] allows to derive initial algebras $\mu(TF)$ from $\mu\,[\![f]\!]$ and $\mu(T^2F)$ from $\mu\,[\![I \times f]\!]$. Note T is a cartesian monad (T, η, θ). Having "rolled around" Δ_I, the two maps above are now given functorially: we define the candidate carrier X for μF simply by the following (coreflexive) equalizer diagram:[4]

$$X \dashrightarrow \mu(TF) \underset{\mu(\eta TF)}{\overset{\mu(T\eta F)}{\underset{\longleftarrow}{\overset{\longrightarrow}{\longleftarrow \mu(\theta F) \longrightarrow}}}} \mu(T^2F)$$

Observe that this is in complete analogy to the (coreflexive) equalizer diagram

$$F \xrightarrow{\eta F} TF \underset{\eta TF}{\overset{T\eta F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\longleftarrow \theta F \longrightarrow}}}} T^2F$$

induced by T since it is cartesian. The algebra structure $h : F(X) \to X$ is induced by coreflexive equalizers being cosifted limits together with preservation properties making the bifunctor $(H, Y) \mapsto H(Y)$ preserve coreflexive equalizers jointly. Initiality of (X, h) can then be seen to transfer from initiality of $\mu(TF)$ using an abstract fibrational argument.

References

[1] S. Awodey, N. Gambino, and K. Sojakova. Inductive types in homotopy type theory. In *Proc. of LICS '12*, pp. 95–104. IEEE CS, 2012.

[2] R. Backhouse, M. Bijsterveld, R. van Geldrop, and J. Van Der Woude. Category theory as coherently constructive lattice theory. Working document, 1998.

[3] N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *Revised Selected Papers from TYPES 2003*, v. 3085 of *Lect. Notes in Comput. Sci.*, pp. 210–225. Springer, 2004.

[4] N. Ghani, P. Hancock, C. McBride, and P. Morris. Indexed containers. Journal version draft, 2013.

[5] K. Szumiło. Two models for the homotopy theory of cocomplete homotopy theories. arXiv:1411.0303, 2014.

[4] It is an artifact of the 1-categorical setting that coreflexive equalizers are special cases of equalizers. In the higher dimensional context, they represent entirely different concepts. But even in the 1-categorical setting, coreflexive equalizers are set apart by being cosifted limits as used above. This confusion might have led to the unnecessary assumption of uniqueness of identity proofs (UIP) in certain formalizations [4].

Axiomatic Set Theory in Type Theory

Gert Smolka

Dept. of Computer Science, Universität des Saarlandes, Germany, smolka@ps.uni-saarland.de

How should we teach axiomatic set theory to students familiar with type theory and proof assistants? Following one of the many textbooks introducing axiomatic set theory does not make sense. In contrast to students of mathematics, our students are familiar with automatic proof checking and an expressive higher-order language for studying axiomatizations. There is no need to talk about the foundations of mathematics. We can just write down the axioms and start proving interesting consequences.

We assume a type theory with excluded middle and an impredicative universe of propositions. We are using the proof assistant Coq. We profit much from Coq's support for inductive definitions und inductive proofs. Given that our inductive definitions concern only predicates, they could be replaced with impredicative definitions.

Let us state the axioms for sets. We assume a type $\mathbf{S}$ and call the members of $\mathbf{S}$ *sets*. A *class* is a predicate $\mathbf{S} \to \text{Prop}$. The letters x, y, and z will range overs sets, and p and q will range over classes. *Inclusion* $p \subseteq q$ and *equivalence* $p \equiv q$ of classes are defined as one would expect. A class is *unique* if it contains at most one set.

We assume a predicate $\in : \mathbf{S} \to \mathbf{S} \to \text{Prop}$ for *set membership*. Notationally, we identify a set x with the class $\lambda z.\, z \in x$. This way notations like $x \subseteq y$, $x \subseteq p$, and $p \equiv x$ are available. A class p is *realizable* if there exists a set x such that $p \equiv x$. It is straightforward to see that the class $\lambda x.\, x \notin x$ is unrealizable. We assume extensionality of sets:

$$x = y \;\leftrightarrow\; x \equiv y$$

We assume constants $\emptyset$, $\{x, y\}$, $\bigcup x$, $\mathscr{P}x$, and $R@x$ to account for the empty set, unordered pairs, unions, power sets, and replacements. The meaning of the constants is given by the following universally quantified axioms:

$$\begin{aligned} z \in \emptyset &\;\leftrightarrow\; \bot \\ z \in \{x, y\} &\;\leftrightarrow\; z = x \vee z = y \\ z \in \textstyle\bigcup x &\;\leftrightarrow\; \exists y \in x.\; z \in y \\ z \in \mathscr{P}x &\;\leftrightarrow\; z \subseteq x \\ z \in R@x &\;\leftrightarrow\; \exists y \in x.\; Ryz \wedge Ry \text{ unique} \end{aligned}$$

The axioms are known from the set theory ZF. The axiom for replacement $R@x$ is higher-order since it quantifies over a relation $R : \mathbf{S} \to \mathbf{S} \to \text{Prop}$. In first-order logic, replacement can only be expressed with a scheme describing infinitely many axioms. Our formulation of the replacement axiom is equivalent to a more conventional formulation requiring R to be functional.

For the results we claim in this abstract we do not need the axioms for infinity, choice, and regularity. For the axioms given one can construct a model in type theory based on Ackermann's encoding [1] of hereditarily finite sets. Thus the axioms given for sets do not affect consistency. For the general case with infinity and choice, consistency has been studied by Aczel [3] and Werner [4].

Singletons and binary unions of sets can be expressed as one would expect: $\{x\} := \{x, x\}$ and $x \cup y := \bigcup\{x, y\}$. Operators for separation $x \cap p$ and description $\ulcorner p \urcorner$ (obtaining the element of a singleton class) can be expressed with replacement.

We define the class $\mathscr{W}$ of *well-founded sets* inductively with a single rule:

$$\frac{x \subseteq \mathscr{W}}{x \in \mathscr{W}}$$

This kind of definition is familiar in type theory but unknown in first-order presentations of set theory. Note that $\mathscr{W}$ defines well-founded sets as sets allowing for epsilon induction (a notion known in first-order set theory).

The regularity axiom says that every set is well-founded. While it is easy to express the regularity axiom as a first-order formula, expressing well-foundedness of a single set is difficult; it seems that the infinity axiom is needed so that transitive closure of sets can be expressed. The limitations of first-order logic may be the reason that ZF comes with the regularity axiom disallowing non-well-founded sets. There is agreement that the regularity axiom is not needed mathematically. There is also Aczel's non-well-founded set theory [2] that has as an axiom postulating the existence of non-well-founded sets.

Next we define that class $\mathscr{Z}$ of *cumulative sets* inductively by means of two rules:

$$\frac{x \subseteq \mathscr{Z}}{\bigcup x \in \mathscr{Z}} \qquad\qquad \frac{x \in \mathscr{Z}}{\mathscr{P}x \in \mathscr{Z}}$$

The class $\mathscr{Z}$ is known as *cumulative hierarchy* or as *Zermelo hierarchy* or as *von Neumann hierarchy* in the literature. In the literature, $\mathscr{Z}$ is defined by transfinite recursion on ordinals. We have not seen a definition of $\mathscr{Z}$ not making use of the ordinals. Here are the main results we have shown for $\mathscr{W}$ and $\mathscr{Z}$:

1. $\mathscr{W}$ and $\mathscr{Z}$ are unrealizable.
2. $\mathscr{W} \equiv \bigcup \mathscr{Z}$.
3. $\mathscr{Z}$ is well-ordered by set inclusion.
4. For every well-ordered set x there exists a unique cumulative set z such that x and the set $\{\, y \in \mathscr{Z} \mid y \subset z \,\}$ are order isomorphic ($\{\, y \in \mathscr{Z} \mid y \subset z \,\}$ is ordered by set inclusion).

Result (4) says that the cumulative sets can serve as unique set representations of the isomorphism classes of well-ordered sets. In the literature, von Neumann ordinals are introduced for this purpose.

The theory of well-orderings and transfinite recursion should be developed generally in pure type theory. It can then be applied to the axiomatized system of sets. For the results mentioned in this abstract neither transfinite recursion nor ordinals are needed.

The Coq development accompanying this abstract has been carried out by Dominik Kirst. Together with a full paper it can be found at `https://www.ps.uni-saarland.de/extras/types15`.

References

[1] W. Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlehre. *Math. Ann.*, 114(1):305–315, 1937.

[2] P. Aczel. *Non-well-founded sets.* v. 14 of *CSLI Lect. Notes.* CSLI, 1988.

[3] P. Aczel. On relating type theories and set theories. In *Selected Papers from TYPES '98*, v. of 1657 of *Lect. Notes in Comput. Sci.*, pp. 1–18. Springer, 1998.

[4] B. Werner. Sets in types, types in sets. In *Proc. of TACS '97*, v. 1281 of *Lect. Notes in Comput. Sci.*, pp. 530–346. Springer, 1997.

Cubical Sets as a Classifying Topos*

Bas Spitters

Carnegie Mellon University, USA / Aarhus University, Denmark
bawspitters@gmail.com

Abstract

Coquand's cubical set model for homotopy type theory provides the basis for a computational interpretation of the univalence axiom and some higher inductive types, as implemented in the *cubical* proof assistant. We show that the underlying cube category is the opposite of the Lawvere theory of De Morgan algebras. The topos of cubical sets itself classifies the theory of 'free De Morgan algebras'. This provides us with a topos with an internal 'interval'. Using this interval we construct a model of type theory following van den Berg and Garner. We are currently investigating the precise relation with Coquand's. We do not exclude that the interval can also be used to construct other models.

The topos of cubical sets

Simplicial sets from a standard framework for homotopy theory. The topos of simplicial sets is the classifying topos of the theory of strict linear orders with endpoints. Cubical sets turn out to be more amenable to a constructive treatment of homotopy type theory. Grandis and Mauri [4] describe the classifying theories for several cubical sets without diagonals. We consider the most recent cubical set model [2]. This consists of symmetric cubical sets with connections $(\wedge, \vee)$, reversions $(\neg)$ and diagonals. Let $\mathbb{F}$ be the category of finite sets with all maps. Consider the monad DM on $\mathbb{F}$ which assigns to each finite set F the *finite* free DM-algebra (=De Morgan-algebra) on F. That this set is finite can be seen using the disjunctive normal form. The *cube category* in [2] is the Kleisli category for the monad DM.

Lawvere theory Recall that for each algebraic (=finite product) theory T, the Lawvere theory $C_{fp}[T]$ is the opposite of the category of free finitely generated models. This is the classifying category for T: models of T in any finite product category category E correspond to product-preserving functors $C_{fp}[T] \to E$. The Kleisli category KL_{DM} is precisely the *opposite* of the Lawvere theory for DM-algebras: maps $I \to DM(J)$ are equivalent to DM-maps $DM(I) \to DM(J)$ since each such DM-map is completely determined by its behavior on the atoms, as $DM(I)$ is free.

Classifying topos To obtain the classifying topos for an algebraic theory, we first need to complete with finite limits, i.e. to consider the category C_{fl} as the *opposite* of finitely *presented* DM-algebras. Then $C_{fl}^{op} \to Set$, i.e. functors on finitely presented T-algebras, is the classifying topos. This topos contains a generic T-algebra M. T-algebras in any topos $\mathcal{F}$ correspond to *left exact left adjoint* functors from the classifying topos to $\mathcal{F}$.

Let FG be the category of *free finitely generated* DM-algebras and let FP the category of *finitely presented* ones. We have a fully faithful functor $f : FG \to FP$. This gives a geometric morphism ϕ between the functor toposes. Since f is fully faithful, ϕ is an embedding.

*I gratefully acknowledge the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

The subtopos Set^{FG} of the classifying topos for DM-algebras is given by a quotient theory, the theory of the model $\phi^* M$. This model is given by pullback and thus is equivalent to the canonical DM-algebra $\mathbb{I}(m) := m$ for each $m \in FG$. So cubical sets are the classifying topos for 'free DM-algebras'. Each finitely generated DM-algebra has the disjunction property and is strict, $0 \neq 1$. These properties are geometric and hence also hold for $\mathbb{I}$. This disjunction property is important in the implementation [2, 3.1].

This result can be generalized to related algebraic structures, e.g. Kleene algebras. A Kleene algebra is a DM-algebra with the property for all x, y, $x \wedge \neg x \leq y \vee \neg y$. With Coquand we checked that free finitely generated Kleene algebras also have the disjunction property.

Model of type theory

Coquand's presentation of the cubical model does not build on a general categorical framework for constructing models of type theory. Docherty [3] presents a model on cubical sets with connections using the general theory of path object categories [5]. The precise relation with the model in [1] is left open. We present a slightly different construction using similar tools, but combined with internal reasoning. We use $\mathbb{I}$ as an 'interval'. To obtain a model of type theory on a category C it suffices to provide an involutive 'Moore path' category object on C with certain properties. Now, category objects on cubical sets are categories in that topos. The Moore path category MX consists of lists of composable paths $\mathbb{I} \to X$ with the zero-length paths e_x as left and right identity. To obtain a *nice* path object category, we quotient by the relation which identifies constant paths of any length. The reversion $\neg$ on $\mathbb{I}$ allows us to reverse paths of length 1. This reversion extends to paths of any length. We obtain an involutive category: Moore paths provide strictly associative composition, but non-strict inverses.

A path contraction is a map $MX \to MMX$ which maps a path p to a path from p to the constant path on tp (t for target). Like Docherty, we use connections to first define the map from $X^{\mathbb{I}}$ to $X^{\mathbb{I}\times\mathbb{I}}$ by $\lambda p.\lambda ij.p(i \vee j)$ and then extended this to a contraction. All these constructions are algebraic and hence work functorially. We obtain a nice path object category.

We have obtained a model of type theory [5, 3] starting from $\mathbb{I}$ in the cubical model. We plan to compare this more carefully with the one in [2]. Like Voevodsky [6], we define intensional identity types inside the extensional type theory of a topos.

Acknowledgements I would like to thank Steve Awodey and Thierry Coquand for discussions on the topic of this paper. Independently, Awodey showed that Cartesian cubical sets (without connections or reversions) classify strictly bipointed objects.

References

[1] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In *Proc. of TYPES 2013*, v. 26 of *Leibniz Int. Proc. in Inform.*, pp. 107–128. Dagstuhl, 2014.

[2] T. Coquand. Lecture notes on cubical sets. Notes, 2015.

[3] S. Docherty. A model of type theory in cubical sets with connections. Master's thesis, University of Amsterdam, 2014.

[4] M. Grandis and L. Mauri. Cubical sets and their site. *Theory and Appl. of Categ*, 11(8):185–211, 2003.

[5] B. van den Berg and R. Garner. Topological and simplicial models of identity types. *ACM Trans. on Comput. Log.*, 13(1), art. 3, 2012.

[6] V. Voevodsky. A type system with two kinds of identity types. Unpublished note, 2013.

Total (Co)Programming with Guarded Recursion

Andrea Vezzosi

Dept. of Comput. Sci. and Engin., Chalmers University of Technology, Sweden
vezzosi@chalmers.se

The standard way to ensure the totality of recursive definitions in systems like Coq or Agda relies on syntactic checks that mainly ensure that the recursive calls are made on structurally smaller arguments. However, such syntactic checks get in the way of code reuse and compositionality. This is especially the case when using higher-order combinators, because they can appear between a recursive call and the actual data it will process. The following definition of a map function for rose trees is an example of this.

$$\begin{aligned}&\textbf{data}\ \mathit{RoseTree}\ A\ =\ \mathit{Node}\ A\ (\mathit{List}\ (\mathit{RoseTree}\ A))\\&\mathit{mapRT} : (A\ \to\ B)\ \to\ \mathit{RoseTree}\ A\ \to\ \mathit{RoseTree}\ B\\&\mathit{mapRT}\ f\ (\mathit{Node}\ a\ ts)\ =\ \mathit{Node}\ (f\ a)\ (\mathit{map}\ (\lambda\ t\ \to\ \mathit{mapRT}\ f\ t)\ ts)\end{aligned}$$

The definition of *mapRT* is not accepted because syntactically t has no relation to *Node a ts*: we need to pay attention to the semantics of *map* to accept this definition as terminating. In fact a common workaround is to essentially inline the code for *map*[1]. Such a system then actively fights abstraction, and offers few recourses other than sticking to some specific syntactic forms for (co)recursive definitions.

Recently there has been a fair amount of research into moving the information about how functions consume and produce data to the type level, so that totality checking can be more modular [3, 5, 1]. In particular, the previous work on Guarded Recursion [3, 5] has handled the case of ensuring totality for corecursion, i.e. manipulating infinite data. The issue of ensuring totality in a modular way for recursion, over well-founded data, was however left open. We address that problem by presenting a calculus that supports both corecursion and recursion with a single guarded fixed point combinator.

$$\frac{\Gamma, i :\ \mathit{Time} \vdash A(i) :\ \mathit{Type} \qquad \Gamma \vdash f : \forall\ i\ .\ (\forall\ j < i\ .\ A(j))\ \to\ A(i)}{\Gamma \vdash \mathit{fix}\ f : \forall\ i\ .\ A(i)}$$

The guarded fixpoint *fix* is presented here as well-founded recursion on the abstract type *Time*. Our bounded quantification $\forall\ j < i$ takes the role of the delay modality of previous works. The name *Time* comes from thinking of such values as representing the amount of time left to perform our computation.

Taking a fixed point on an universe U we can define both guarded and standard coinductive streams.

$$\begin{aligned}&\mathit{gStream}\ A\ =\ \mathit{fix}\ (\lambda\ i.\ \lambda\ (X : \forall\ j < i.\ U)\ .\ A \times (\forall\ j < i\ .\ X\ j))\\&\mathit{Stream}\ A\ =\ \forall\ i.\ \mathit{gStream}\ A\ i\end{aligned}$$

The induction hypothesis only provides an element of U when given a smaller *Time*, however we want to guard recursive occurrences with $\forall\ j < i$, so we have the smaller time j available.

[1] If the definition of *map* is available, Coq will attempt this automatically.

The type $\forall\ i.\ gStream\ A\ i$ then corresponds to the standard coinductive stream type, since we can choose an i big enough to inspect the stream as deeply as we need.

The above technique reproduces previous results [3, 5], however in our language we also obtain inductive types, by making use of the existential rather than universal quantification over *Time*.

$$\begin{aligned} gNat &= fix\ (\lambda\ i\ (X : \forall\ j < i.\ U).\ \top + (\exists\ j < i.\ X\ j)) \\ Nat &= \exists\ i.\ gNat\ i \end{aligned}$$

An element of $gNat\ i$ is a Peano natural bounded in height by i. An element of *Nat* is then an arbitrary natural number since we can pick a large enough bound.

It is however not enough to quantify over the time to get the right type, for example we risk having too many representations of 0 if we can tell the difference between $(i, inl\ tt)$ and $(j, inl\ tt)$ for two different times i and j.

The key idea is that values of type $\exists\ i.\ A$ must keep abstract the specific time they were built with, exactly like weak sums in System F. Intuitively Nat will not be the initial algebra of $(\top +)$ unless $Nat \cong \top + Nat$ holds, so *Nat* must be able to support both an interface and an equational theory where times play no role.

We characterize this invariance by a suitable interpretation of *Time* and the time quantifiers in the relational parametric model of type theory of [2][2]. In particular while $\forall\ i$ is just a Π type, the existential quantification is not simply a Σ type but requires an interpretation analogous to the one of the existential types of the polymorphic lambda calculus [4]. In the calculus we provisionally internalize this invariance as type isomorphisms. In the specific case of *Nat*, both $(i, inl\ tt)$ and $(j, inl\ tt)$ get sent to $inl\ tt$ by the isomorphism, so we can conclude they are equal.

An example that highlights the expressivity of the resulting system is the ability to give a safe type to the *unfold* combinator for lists:

$$\begin{aligned} &unfold : (\forall\ i.\ S\ i\ \to\ \top + (A \times \exists\ j < i.\ S\ j))\ \to\ \forall\ i.\ S\ i\ \to\ List\ A \\ &unfold\ f\ =\ fix\ \lambda\ i\ unfold'\ s.\ case\ f\ i\ s\ of \\ &\qquad\qquad\qquad\qquad\qquad\qquad inl\ \_\ \to\ [\,] \\ &\qquad\qquad\qquad\qquad\qquad\qquad inr\ (a, (j, s'))\ \to\ a :: unfold'\ j\ s' \end{aligned}$$

References

[1] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In *Proc. of ICFP '13*, pp. 185–196. ACM, 2013.

[2] R. Atkey, N. Ghani, and P. Johann. A relationally parametric model of dependent type theory. In *Proc. of POPL '14*, pp. 503–516. ACM, 2014.

[3] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *Proc. of ICFP '13*, pp. 197–208. ACM, 2013.

[4] B. P. Dunphy and U. S. Reddy. Parametric limits. In *Proc. of LICS '04*, pp. 242–251. IEEE CS, 2004.

[5] R. E. Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Proc. of CSL-LICS 2014*, art. 71. ACM, 2014.

[2]This gives a sound denotational model for our calculus and we are currently working on deriving operational semantics that enjoy strong normalization and decidable typechecking by constraining the unfolding of the *fix* operator.

Balanced Polymorphism and Linear Lambda Calculus

Noam Zeilberger

Microsoft Research – INRIA Joint Centre, France
noam.zeilberger@gmail.com

Abstract

A "pearl theorem" of lambda calculus says that every linear lambda term is simply typable, and moreover that its $\beta\eta$-normal form is uniquely determined by its principal type. Mints (1977) gave a proof-theoretic demonstration of this result, by showing that 1) every linear lambda term has a *balanced* principal typing sequent (where a sequent $\Gamma \to A$ is said to be balanced if each atomic formula occurring in it has exactly two occurrences, once in positive position and once in negative position), and 2) any balanced sequent is inhabited by at most one $\beta\eta$-normal form. We give a new more conceptual proof of Mints' pearl theorem, describing it first as a simple bijection between string diagrams for linear normal forms and provable balanced sequents, and second as a simple bidirectional type inference algorithm for linear lambda terms (that is exactly dual to standard bidirectional type checking).

Linear lambda calculus represents an extremal case of parametricity, in the sense that every linear lambda term is typable, and its normal form is uniquely identified by its principal type. For example, the normal forms on the left are uniquely identified by the types on the right, and vice versa:

$$\begin{array}{rcc} \lambda x.x(\lambda y.y) & : & ((\alpha \multimap \alpha) \multimap \beta) \multimap \beta \\ \lambda x.\lambda y.xy & : & (\alpha \multimap \beta) \multimap (\alpha \multimap \beta) \\ \lambda x.\lambda y.y(x(\lambda z.z)) & : & ((\alpha \multimap \alpha) \multimap \beta) \multimap ((\beta \multimap \gamma) \multimap \gamma) \end{array}$$

One consequence of this observation is that type inference is essentially equivalent to normalization, and it is in that context that I first saw this property asserted as a "pearl theorem" by Mairson [1]. The result should probably be attributed to Mints [2], who proved a version of it in the context of the so-called coherence problem for monoidal closed categories (also studied notably by Lambek, and by Kelly and Mac Lane). Mints' proof can be divided into the following pair of observations:

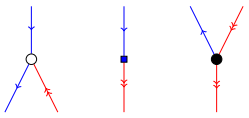
1. The principal type of a linear lambda term is *balanced*, in the sense that every type variable occurring in it occurs exactly twice, once positively and once negatively. For example, here I have colored in red and blue the positive and negative occurrences of the three type variables in the principal type of $\lambda x.\lambda y.y(x(\lambda z.z))$: $((\alpha \multimap \alpha) \multimap \beta) \multimap (\beta \multimap \gamma) \multimap \gamma$
2. Any balanced type (and more generally, any balanced typing sequent) has at most one inhabitant up to $\beta\eta$-equivalence.

Both (1) and (2) are established by relatively straightforward proof-theoretic arguments (applying an induction on the length of terms).

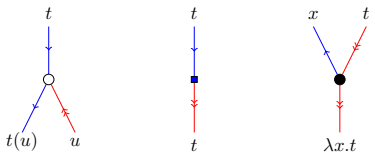
Although the proof in [2] is not very complicated, a pearl theorem deserves a pearl proof.[1] Our starting point will be a simple string diagram representation of *normal* linear lambda terms

[1] There are other reasons why I think that the "balanced polymorphism" exhibited in linear lambda calculus is worth being studied on its own terms. For example, it corresponds closely to the notion of "end" in category theory.

that was introduced in [4] as a refinement of so-called lambda-graphs, and which is derived from the refinement type signature [3] encoding the well-known characterization of normal lambda terms in mutual induction with "neutral" terms. In this representation, normal and neutral terms correspond to certain diagrams constructed using three basic combinators which we call a-, s-, and ℓ-nodes (standing for $a$pplication, the $s$witch from neutral to normal, and ℓambda abstraction) on colored, oriented wires:



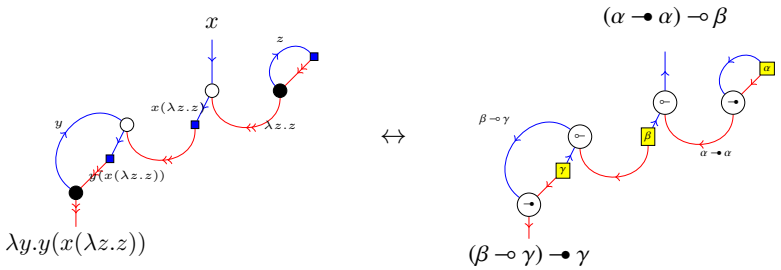
These components may be annotated like so (note that blue wires carry neutral terms and red wires carry normal terms):



From the string diagram of a normal lambda term, you obtain a balanced typing sequent by the following procedure:

1. reverse the orientation of blue (neutral) wires
2. turn each a-node into a negative-polarity implication (i.e., an implication occurring in negative position), and each ℓ-node into a positive-polarity implication
3. replace each s-node by a distinct type variable (with two outgoing wires, standing for its positive and negative occurrences)
4. read off a balanced principal typing sequent for the original term by starting at the type variables and pushing information along the direction of the wires

For example, here is the transformation applied on the diagram of the closed normal term $\lambda x.\lambda y.y(x(\lambda z.z))$ (with its outermost λx removed to view it as a term with one free variable):



Moreover, this transformation is clearly reversible (since the choice of type variable names is irrelevant). Finally, we can turn this into a traditional type inference algorithm (much simpler than Hindley-Milner inference on this very special case), described in the following rules:

$$\frac{}{x : \alpha \Leftarrow x \Leftarrow \alpha} \qquad \frac{\Gamma \Leftarrow t \Leftarrow A \multimap B \quad \Delta \Leftarrow u \Rightarrow A}{\Gamma, \Delta \Leftarrow t(u) \Leftarrow B} \qquad \frac{\Gamma \Leftarrow t \Leftarrow \alpha \quad \alpha \text{ fresh}}{\Gamma \Leftarrow t \Rightarrow \alpha} \qquad \frac{x : A, \Gamma \Leftarrow t \Rightarrow B}{\Gamma \Leftarrow \lambda x.t \Rightarrow A \multimap B}$$

Here the two judgment forms are $\Gamma \Leftarrow t \Leftarrow A$ ("checking against A, t synthesizes context Γ") and $\Gamma \Leftarrow t \Rightarrow A$ ("t synthesizes context Γ and type A"). The reader familiar with standard bidirectional type checking (where neutral terms synthesize and normal terms check) will remark that this system is exactly dual (suggesting that we might call it "bidirectional type inference").

References

[1] H. G. Mairson. Linear lambda calculus and PTIME-completeness. *J. of Funct. Program.*, 14(6):623–633, 2004.

[2] G. E. Mints. Closed categories and the theory of proofs. *Zapiski Nauchnykh Seminarov LOMI im. V.A. Steklova AN SSSR*, 68:83–114, 1977. Translation in *J. of Soviet Math.*, 15(1):45–62, 1981, reprinted in G. E. Mints, *Selected Papers in Proof Theory*, Bibliopolis, 1992.

[3] F. Pfenning. Refinement types for logical frameworks. In *Informal Proc. of TYPES '93*, pp. 315–328, 1993. `http://www.cse.chalmers.se/research/group/logic/Types/proc93.ps`

[4] N. Zeilberger and A. Giorgetti. A correspondence between rooted planar maps and normal planar lambda terms. *Log. Methods in Comput. Sci.*, to appear. arXiv:1408.5028

Author index