TYPES 2021 – Book of Abstracts

June 2021

The TYPES meeting series has always been my favourite among all conferences and I am honoured that I was given the opportunity to organise TYPES 2021. It was, unfortunately, during the years 2020 and 2021 not possible to hold any physical conferences. Ugo and Stefano had done great work in the preparation of TYPES 2020, but the sudden disruption made it impossible to hold the meeting. This year, we were able to prepare ourselves to hold TYPES 2021 as a virtual meeting. Even though this is far from ideal for a meeting that is centred around collaboration, it was still very enjoyable and we had many very high quality submissions, as this book of abstracts attests. I wish to thank all authors and participants for making TYPES 2021 such a rich and enjoyable meeting, despite the circumstances. Enjoy the abstracts!

Henning Basold

Programme Committee

- Andreas Abel (Gothenburg University/Chalmers)
- Henning Basold (LIACS — Leiden University) (chair)
- Stefano Berardi (Università di Torino)
- Marc Bezem (University of Bergen)
- Frédéric Blanqui (INRIA — ENS Paris-Saclay)
- Sandrine Blazy (INRIA — University of Rennes)
- Ana Bove (Chalmers / U. of Gothenburg)
- Paolo Capriotti (no affiliation)
- Jesper Cockx (TU Delft)
- José Espírito Santo (University of Minho)
- Herman Geuvers (Radboud University Nijmegen)
- Silvia Ghilezan (University of Novi Sad)
- Nicolai Kraus (U. of Nottingham)
- Sergueï Lenglet (Univ. de Lorraine)
- Assia Mahboubi (INRIA — Nantes and VU Amsterdam)
- Ralph Matthes (IRIT — CNRS and University of Toulouse)
- Keiko Nakata (SAP Postdam)
- Fredrik Nordvall Forsberg (University of Strathclyde)
- Jorge A. Pérez (University of Groningen)
- Gabriel Scherer (INRIA Saclay)
- Aleksy Schubert (University of Warsaw)
- Niccolò Veltri (Tallin UT)
- Stephanie Weirich (U. of Pennsylvania)

Invited Talks

Stephanie Balzer – Session Logical Relations for Noninterference

In this talk I introduce the audience to linear session types through the lens of noninterference. Session types, as the types of message-passing concurrency, naturally capture what information is learned by the exchange of messages, facilitating the development of a flow-sensitive information flow control (IFC) type system guaranteeing noninterference. Noninterference ensures that an observer (adversary) cannot infer any secrets from made observations. I will explain the key ideas underlying the development of the IFC type system as well as the construction of the logical relation conceived to prove noninterference. The type system is based on intuitionistic linear logic and enriched with possible worlds to impose invariants on run-time configurations of processes, leading to a stratification in line with the security lattice. The logical relation generalizes existing developments for session-typed languages to open configuration to allow for a more subtle statement of program equivalence.

Ulrik Buchholtz – Genuine pairs and the trouble with triples in homotopy type theory

What is an unordered pair of groups, or of categories, or of types? In homotopy type theory, the naive answer is a pair of a two-element type K and a function from K to whatever we're taking pairs of. This definition has the downside that the type of unordered pairs in a set is no longer a set. For example, the unordered pairs in the unit type is the infinite dimensional real projective space.

I'll explain how to define genuine unordered pairs using ideas from equivariant homotopy theory and prove that the genuine unordered pairs in a set is still a set. Then I'll define genuine unordered triples and relate my troubles with them. Finally, I'll explain why genuine multisets of arbitrary cardinality are really difficult.

Sara Negri – On the constructive content of infinitary classical theories

Notable parts of algebra and geometry can be formalized as coherent theories over first-order logic. Albeit wide, the class of coherent theories misses certain axioms in algebra such as the axioms of torsion abelian groups, Archimedean ordered fields, or the notion of transitive closure used in the theory of connected graphs and in the modelling of epistemic social notions such as common knowledge. All those examples can however be axiomatized by means of geometric axioms, a generalization of coherent axioms that includes infinitary disjunctions. We give a constructive proof of cut elimination for infinitary classical and intuitionistic sequent calculi for geometric theories. We then exploit their uniformity to obtain, purely by methods of structural proof theory, Glivenko-style conservativity results leading to the infinitary generalization of the first-order Barr theorem as a special case.

(Includes joint work with Giulio Fellin and Eugenio Orlandelli)

Pierre-Marie Pédrot – All your base categories are belong to us: A syntactic model of presheaves in type theory

Presheaves are a staple categorical structure, which naturally arises in a wide variety of situations. In the realm of logic, they are often used as a model factory. Indeed, presheaves over some base category will result in a topos, whose contents can be fine-tuned by carefully picking the base category. As computer scientists, though, we have learnt that there are even better logical systems than toposes: dependent type theories! Through the Curry-Howard mirror, they are also full-blown functional programming languages that actually compute.

This begs the following question: is it possible to build the type-theoretic equivalent of presheaves, while retaining the good computational properties of our dependent programming languages? We will see that strikingly enough, presheaves can already be presented as computational objects to some extent, except for the annoying fact that they do not obey the right conversion rules! A proper account of type-theoretic presheaves will require a coming-of-age journey through the world of effectful program semantics, using fine and modern tools such as call-by-push-value, dependent parametricity and strict equality. In the end, we will formulate an alternative presentation of presheaves in type theory, but which is still equivalent to its standard categorical counterpart when viewed from the static world of sets. As an application, we will use them to extend dependent type theory with new effective logical principles.

Contributed Talks

On Model-Theoretic Strong Normalization for Truth-Table Natural Deduction

Andreas Abel

Department of Computer Science and Engineering, Gothenburg University

Geuvers and Hurkens [2017a] introduced a method to derive natural deduction proof rules from truth tables of logical connectives.

For instance, consider the truth table for implication. For each line where $A \to B$ holds, *e.g.*, the second line, an introduction rule is created where 0-valued (or *negative*) operands A become premises $\Gamma.A \vdash A \to B$ and 1-valued (or *positive*) operands B become premises $\Gamma \vdash B$. Lines like the third where $A \to B$ is false become eliminations with a conclusion $\Gamma \vdash C$ for an arbitrary formula C. The premises of this eliminations are a premise $\Gamma \vdash A$ for each 1-valued operand A, and a premise $\Gamma.B \vdash C$ for each 0-valued operand B. This yields the following four rules for judgements $\Gamma \vdash A$ and $\Gamma \mid A \vdash C$, which we complement by the standard administrative rules for natural deduction in spine form:[1]

A	B	$A \to B$
0	0	1
0	1	1
1	0	0
1	1	1

$$\mathsf{in}^{00}_{\to}\ \frac{\Gamma.A \vdash A \to B \qquad \Gamma.B \vdash A \to B}{\Gamma \vdash A \to B} \qquad \mathsf{in}^{01}_{\to}\ \frac{\Gamma.A \vdash A \to B \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

$$\mathsf{el}^{10}_{\to}\ \frac{\Gamma \vdash A \qquad \Gamma.B \vdash C}{\Gamma \mid A \to B \vdash C} \qquad \mathsf{in}^{11}_{\to}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

$$\mathsf{var}\ \frac{A \in \Gamma}{\Gamma \vdash A} \qquad \mathsf{elim}\ \frac{\Gamma \vdash A \qquad \Gamma \mid A \vdash C}{\Gamma \vdash C} \qquad \mathsf{id}\ \frac{}{\Gamma \mid A \vdash A} \qquad \mathsf{comp}\ \frac{\Gamma \mid A \vdash B \qquad \Gamma \mid B \vdash C}{\Gamma \mid A \vdash C}$$

The rule names serve as the constructors for proof terms $a, b, c, f, t, u : \Gamma \vdash A$. Exceptions are elim and comp which we write infix as centered dot $f \cdot \vec{E}$ and $E \cdot \vec{E}$, and var which we omit, treating an index $x : A \in \Gamma$ directly as proof term. Further, we consider eliminations $\vec{E} : \Gamma \mid A \vdash C$ up to associativity of composition with identity id. Substitution of the last hypothesis in $t : \Gamma.A \vdash C$ by $a : \Gamma \vdash A$ is written $t[a] : \Gamma \vdash C$.

Proof terms allow us to express the reduction rules concisely. A *detour* or β-reduction can fire on the elimination $I \cdot E$ of an introduction I.

- Either, a positive premise (1) of the introduction matches a negative premise (0) of the elimination. E.g., the second premise of the elimination $\mathsf{el}^{10}_{\to}$ is negative, and it can react with the positive second premise of $\mathsf{in}^{01}_{\to}$ and $\mathsf{in}^{11}_{\to}$:

$$\mathsf{in}^{1}_{\to}(\_, b) \cdot \mathsf{el}^{10}_{\to}(\_, t) \quad \mapsto_\beta \quad t[b]$$

[1] Geuvers and Hurkens describe the rules in conventional natural deduction format where the elimination rule has the eliminatee as premise. In contrast, we introduce a separate syntactic class $E : \Gamma \mid A \vdash C$ of eliminations that is often associated with sequent calculus. However, with its focus on eliminatee A this is rather a variant of natural deduction where subsequent eliminations can be grouped together. For us, this is more of a convenience of notation and presentation than a deviation from Geuvers and Hurkens. Our arguments all hold also with the original natural deduction rules.

- Or a negative premise of the introduction, e.g., $\mathsf{in}^{00}_{\to}$ or $\mathsf{in}^{01}_{\to}$, reacts with a matching positive premise of the elimination, $\mathsf{el}^{10}_{\to}$. In this case, the elimination persists, but the introduction is replaced with an instantiation of its respective negative premise.

$$\mathsf{in}^{0\_}_{\to}(u, \_) \cdot \mathsf{el}^{10}_{\to}(a, t) \quad \mapsto_\beta \quad u[a] \cdot \mathsf{el}^{10}_{\to}(a, t)$$

Reduction is inherently non-confluent: the reducts of $\mathsf{in}^{01}_{\to}(u, b) \cdot \mathsf{el}^{10}_{\to}(a, t)$ form the critical pair of $t[b]$ and $u[a] \cdot \mathsf{el}^{10}_{\to}(a, t)$ which can in general not be joined.

Permutation or π-reductions allow us to combine sequences of eliminations $E_1 \cdot E_2$ into a single elimination $E_1\{E_2\}$, permuting elimination E_2 into the negative branches of E_1. For implication $A \to B$, we have $\mathsf{el}^{10}_{\to}(a, u)\{E_2\} = \mathsf{el}^{10}_{\to}(a, u \cdot E_2\uparrow)$ where $\uparrow$ lifts $E_2 : \Gamma \mid C \to C' \vdash D$ under the extra hypothesis B of $u : \Gamma.B \vdash C \to C'$.

Geuvers, van der Giessen, and Hurkens [2019] prove strong normalization (SN) of $\beta\pi$-reduction by a translation to simply-typed parallel lambda-calculus. In a forth-coming article [Abel, 2021] in the TYPES 2020 post-proceedings I obtain this result by adapting established model-theoretic SN proofs.

In the adaptation of the (bi)orthogonality method I model types A as sets $\mathcal{A}$ of eliminations $\vec{E}$ of type A containing id. Terms a of type A are then shown to be *orthgonal* to $\mathcal{A}$, where $a \perp \mathcal{A}$ means that $a \cdot \vec{E}$ is SN for all $\vec{E} \in \mathcal{A}$. With notation $\vec{E} \in \mathcal{X}[\mathcal{A}]$ whenever $\vec{E} : \Gamma.A \mid X \vdash C$ and $\vec{E}[a] \in \mathcal{X}$ for all $a \perp \mathcal{A}$, we can construct $\mathcal{A} \to \mathcal{B}$ as the greatest fixpoint $\nu\mathcal{F}^\perp$ of the pointwise orthogonal $\mathcal{F}^\perp$ of the operator

$$\mathcal{F}(\mathcal{X})_\Gamma = \{\mathsf{in}^{00}_{\to}(t, u), \mathsf{in}^{01}_{\to}(t, b), \mathsf{in}^{11}_{\to}(a, b) \mid a \perp \mathcal{A}_\Gamma, b \perp \mathcal{B}_\Gamma, t \perp \mathcal{X}[\mathcal{A}]_\Gamma, u \perp \mathcal{X}[\mathcal{B}]_\Gamma\}.$$

While $\mathcal{F}^\perp$ is monotone, it is not strictly positive, so we cannot directly implement this construction in Agda or predicative Coq, that only support *strictly* positive coinductive types.

An adaptation of Tait's saturated sets method however allows for a predicate proof using strictly positive inductive types only. The saturation condition for a set $\mathcal{C}$ of SN terms is a mix of conditions presented by Geuvers and Hurkens [2017b, 2020] and Matthes [2005]:

1. $\mathcal{C}$ contains any SN weak head β-redex whose reducts it already contains.
2. $\mathcal{C}$ contains the variables x.
3. $\mathcal{C}$ contains a SN neutral $x \cdot \mathsf{el}^{10}_{\to}(a, u)$ whenever it contains u.
4. $\mathcal{C}$ contains neutral $x \cdot E_1 \cdot E_2 \cdot \vec{E}$ whenever it contains its π-reduct $x \cdot E_1\{E_2\} \cdot \vec{E}$ and $y \cdot E_2 \cdot \vec{E}$ for some variable y.

The closure operator derived from these conditions is strictly positive, so we can define semantics types like $\mathcal{A} \to \mathcal{B}$ in a predicative way, by their introductions and the inductive closure generated from the above conditions.

Having explained the interpretation of types A as SN term set $\mathcal{A}$, the rest of the SN proof is routine. For pedagogical purposes, we have only spelled out rules, reductions, and semantics for *implication*, which is almost the general case—with the exception that it has only one elimination. Rules and reductions for other connectives can be obtained according to the described schema, and the construction of their semantics can be mechanically derived from the inference rules.

Acknowledgments. Thanks to the anonymous referees for their feedback on this abstract. This work was supported by Vetenskapsrådet under Grant No. 2019-04216 *Model Dependent Type Theory*.

References

A. Abel. On model-theoretic strong normalization for truth-table natural deduction. In *Post-proceedings of the 26th International Conference on Types for Proofs and Programs (TYPES 2020)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. http://www.cse.chalmers.se/~abela/types20post.pdf.

H. Geuvers and T. Hurkens. Deriving natural deduction rules from truth tables. In S. Ghosh and S. Prasad, editors, *Logic and Its Applications - 7th Indian Conference, ICLA 2017, Kanpur, India, January 5-7, 2017, Proceedings*, vol. 10119 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2017a. https://doi.org/10.1007/978-3-662-54069-5_10.

H. Geuvers and T. Hurkens. Proof terms for generalized natural deduction. In A. Abel, F. N. Forsberg, and A. Kaposi, editors, *23rd International Conference on Types for Proofs and Programs, TYPES 2017, May 29-June 1, 2017, Budapest, Hungary*, vol. 104 of *Leibniz International Proceedings in Informatics*, pages 3:1–3:39. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017b. https://doi.org/10.4230/LIPIcs.TYPES.2017.3.

H. Geuvers and T. Hurkens. Addendum to "Proof terms for generalized natural deduction". Retrieved 2021-01-13, 2020. http://www.cs.ru.nl/~herman/PUBS/addendum_to_TYPES.pdf.

H. Geuvers, I. van der Giessen, and T. Hurkens. Strong normalization for truth table natural deduction. *Fundamenta Informaticae*, 170(1-3):139–176, 2019. https://doi.org/10.3233/FI-2019-1858.

R. Matthes. Non-strictly positive fixed-points for classical natural deduction. *Annals of Pure and Applied Logic*, 133(1–3):205–230, 2005. https://doi.org/10.1016/j.apal.2004.10.009.

SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq

Carmine Abate[1], Philipp G. Haselwarter[2], Exequiel Rivas[3], Antoine Van Muylder[4], Théo Winterhalter[1], Cătălin Hriţcu[1], Kenji Maillard[5], and Bas Spitters[2]

[1]MPI-SP [2]Aarhus University [3]Inria Paris [4]Vrije Universiteit Brussel [5]Inria Rennes

Abstract

State-separating proofs (SSP) is a recent methodology for structuring game-based cryptographic proofs in a modular way. While very promising, this methodology was previously not fully formalized and came with little tool support. We address this by introducing SSProve, the first general verification framework for machine-checked state-separating proofs. SSProve combines high-level modular proofs about composed protocols, as proposed in SSP, with a probabilistic relational program logic for formalizing the lower-level details, which together enable constructing fully machine-checked crypto proofs in the Coq proof assistant. Moreover, SSProve is itself formalized in Coq, including the algebraic laws of SSP, the soundness of the program logic, and the connection between these two verification styles.

Note. This work has been accepted at the CSF'21 conference. A full length preprint is available on eprint at https://eprint.iacr.org/2021/397.

State Separating Proofs. Cryptographic proofs can be challenging to make fully precise and to rigorously check. This has caused a "crisis of rigor" [7] in cryptography that Shoup [25], Bellare and Rogaway [7], Halevi [15], and others, proposed to address by systematically structuring proofs as sequences of games. This game-based proof methodology is not only ubiquitous in provable cryptography nowadays, but also amenable to full machine-checking in proof assistants such as Coq [2, 20] and Isabelle/HOL [6]. It has also led to the development of specialized proof assistants [4] and automated verification tools for crypto proofs [3, 5, 10]. There are two key ideas behind these tools: (i) formally representing games and the adversaries against them as code in a probabilistic programming language, and (ii) using program verification techniques to conduct all game transformation steps in a machine-checked manner.

For a long time however, game-based proofs have lacked modularity, which made them hard to scale to large, composed protocols such as TLS [23] or the upcoming MLS [1]. To address this issue, Brzuska et al. [11] have recently introduced *state-separating proofs (SSP)*, a methodology for modular game-based proofs, inspired by the paper proofs in the miTLS project [8, 9, 14], by prior compositional cryptography frameworks [12, 18], and by process algebras [19]. In the SSP methodology, the code of cryptographic games is split into packages, which are modules made up of procedures sharing state. Packages can call each other's procedures (also known as oracles) and can operate on their own state, but cannot directly access other packages' state. Packages have natural notions of sequential and parallel composition that satisfy simple algebraic laws, such as associativity of sequential composition. This law is used to define cryptographic reductions not only in SSP, but also in the *The Joy of Cryptography* textbook [24], which teaches crypto proofs in a style very similar to SSP.

While the SSP methodology is very promising, the lack of a complete formalization makes it currently only usable for informal paper proofs, not for machine-checked ones. The SSP paper [11] defines package composition and the syntax of a cryptographic pseudocode language for games and adversaries, but the semantics of this language is not formally defined, and the meaning of their `assert` operator is not even clear, given the probabilistic setting. Moreover, while SSP provides a good way to structure proofs at the high-level, using algebraic laws such as associativity, the low-level details of such proofs are usually treated very casually on paper. Yet none of the existing crypto verification tools that could help machine-check these low-level details supports the high-level part of SSP proofs: equational reasoning about composed packages (i.e., modules) is either not possible at all [2, 15, 20, 26], or does not exactly match the SSP package abstraction [4, 16].

Contributions. The main contribution of this work is to introduce SSProve, the first general verification framework for machine-checked state-separating proofs. SSProve brings together two different proof styles into a single unified framework: (1) high-level proofs are modular, done by reasoning equationally about composed packages, as proposed in SSP [11]; (2) low-level details are formally proved in a probabilistic relational program logic [2, 4, 20]. Importantly, we show a formal connection between these two proof styles.

SSProve is, moreover, a foundational framework, fully formalized itself in Coq. For this we define the syntax of crypto pseudocode in terms of a free monad, in which external calls are represented as algebraic operations [21]. This gives us a principled way to define sequential composition of packages based on an algebraic effect handler [22] and to give machine-checked proofs of the SSP package laws [11], some of which were treated informally on paper. We moreover make precise the minimal state-separation requirements between adversaries and the games with which they are composed—this reduces the proof burden and allows us to prove more meaningful security results, that do not require the adversary's state to be disjoint from intermediate games in the proof.

Beyond just syntax, we also give a denotational semantics to crypto code in terms of stateful probabilistic functions that can signal assertion failures by sampling from the empty probability subdistribution. Finally, we prove the soundness of a probabilistic relational program logic for relating pairs of crypto code fragments.

For this soundness proof we build a semantic model based on relational weakest-precondition specifications. Our model is modular with respect to the considered side-effects (currently probabilities, state, and assertion failures). To obtain it, we follow a general recipe by Maillard et al. [17], who recently proposed to characterize such semantic models as relative monad morphisms, mapping two monadic computations to their canonical relational specification. This allows us to first define a relative monad morphism for probabilistic, potentially failing computations and then to extend this to state by simply applying a relative monad transformer. Working out this instance of Maillard et al.'s [17] recipe involved formalizing various non-standard categorical constructs in Coq, in an order-enriched context: lax functors, lax natural transformations, left relative adjunctions, lax morphisms between such adjunctions, state transformations of such adjunctions, etc. This formalization is of independent interest and should also allow us to more easily add extra side-effects and F*-style sub-effecting [26] to SSProve in the future.

Case studies. To test our methodology, we have formalized several security proofs in SSProve. The first example looks at a symmetric encryption scheme built out of a pseudo-random function. The second example proves security of ElGamal, a popular asymmetric encryption scheme. Finally, we tackle the more challenging KEM-DEM example of [11], showing that composing a secure key-encapsulation mechanism (KEM) with a data encapsulation mechanism (DEM) results in a secure public-key encryption mechanism, following [13].

The full formalization of SSProve and of the examples mentioned above (circa 20k lines of Coq code including comments) is available at `https://github.com/SSProve/ssprove` under the MIT open source license.

Acknowledgements. We are grateful to Arthur Azevedo de Amorim, Théo Laurent, Nikolaj Sidorenco, and Ramkumar Ramachandra for their technical support and for participating in stimulating discussions. This work was in part supported by the European Research Council under ERC Starting Grant SECOMP (715753), by AFOSR grant *Homotopy type theory and probabilistic computation* (12595060), and by the Concordium Blockchain Research Center at Aarhus University. Antoine Van Muylder holds a PhD Fellowship from the Research Foundation – Flanders (FWO).

References

[1] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (MLS) protocol. IETF Draft, 2020.

[2] G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. *POPL*, 2009.

[3] G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, Y. Lakhnech, B. Schmidt, and S. Zanella Béguelin. Fully automated analysis of padding-based encryption in the computational model. In *CCS'13*. 2013.

[4] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 2013.

[5] G. Barthe, B. Grégoire, and B. Schmidt. Automated proofs of pairing-based cryptography. In *CCS'15*. 2015.

[6] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *J. Cryptol.*, 33(2), 2020.

[7] M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptol. ePrint Arch.*, page 331, 2004.

[8] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Zanella Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO'14*. 2014.

[9] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella Béguelin, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *IEEE S&P*, 2017.

[10] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE S&P*. 2006.

[11] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State separation for code-based game-playing proofs. In *ASIACRYPT*. 2018.

[12] R. Canetti. Universally composable security. *J. ACM*, 67(5), 2020.

[13] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 33(1), 2003.

[14] C. Fournet, M. Kohlweiss, and P. Strub. Modular code-based cryptographic verification. *CCS*. 2011.

[15] S. Halevi. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch.*, page 181, 2005.

[16] A. Lochbihler, S. R. Sefidgar, D. A. Basin, and U. Maurer. Formalizing constructive cryptography using CryptHOL. In *CSF*. 2019.

[17] K. Maillard, C. Hriţcu, E. Rivas, and A. V. Muylder. The next 700 relational program logics. *Proc. ACM Program. Lang.*, 4(POPL), 2020.

[18] U. Maurer and R. Renner. Abstract cryptography. In *Innovations in Computer Science - ICS'11*. 2011.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1), 1992.

[20] A. Petcher and G. Morrisett. The foundational cryptography framework. *POST*. 2015.

[21] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1), 2003.

[22] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. *ESOP*. 2009.

[23] E. Rescorla. The transport layer security (TLS) protocol version 1.3. IETF RFC 5246, 2018.

[24] M. Rosulek. The Joy of Cryptography. Online textbook, 2021.

[25] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.*, page 332, 2004.

[26] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

B-systems and C-systems are equivalent

Benedikt Ahrens[1]*, Jacopo Emmenegger[1]*, Paige North[2], and Egbert Rijke[3]†

[1] University of Birmingham, UK, b.ahrens@cs.bham.ac.uk, j.j.emmenegger@bham.ac.uk
[2] University of Pennsylvania, US, pnorth@upenn.edu
[3] University of Ljubljana, Slovenia, egbert.rijke@fmf.uni-lj.si

Abstract

C-systems were defined by Cartmell as models of generalized algebraic theories. B-systems were defined by Voevodsky in his quest to formulate and prove an initiality theorem for type theories. In this work we prove Voevodsky's conjecture that the categories of B-systems and of C-systems are equivalent. We do so by specialising a more general equivalence between non-stratified versions of B-systems and C-systems, which we name E-systems and CE-systems, respectively.

In his unfinished and only partially published [3–7] research on type theories, Voevodsky aimed to develop a mathematical theory of type theories, similar to the theory of groups or rings. In particular, he aimed to state and prove rigorously an "Initiality Conjecture" for type theories, in line with the initial semantics approach to the syntax of (programming) languages.

One aspect of this Initiality Conjecture is to construct, from the types and terms of a programming language, a "model", that is, a mathematical object—typically, a category equipped with some extra structure. To help with this endeavour in the context of initial semantics for type theories, Voevodsky introduced the essentially-algebraic theory of *B-systems*. The models of this theory, he conjectured in [2], are constructively equivalent to the well-known *C-systems* or *contextual categories*, first introduced by Cartmell as a mathematical structure for the interpretation of the rules of type theory [1, § 14]. A C-system is a category coming, in particular, with a length function and a compatible "father" function on objects of the category, signifying truncation of contexts. Furthermore, there is a special class of morphisms $\mathcal{F}$, closed under pullback of arbitrary morphisms—thought of as substitution by that morphism.

Voevodsky's definition of B-systems [2] is inspired by the presentation of type theories in terms of *inference rules*. Specifically, type theories "of Martin-Löf genus" are given by sets of five kinds of judgements, namely well-formed context ($\Gamma \vdash$), well-formed type in some context ($\Gamma \vdash A\,\mathsf{type}$), well-formed term of some type in some context ($\Gamma \vdash a : A$), equality of types ($\Gamma \vdash A \equiv B$) and equality of terms ($\Gamma \vdash a \equiv b : A$). Interpreting equality of types and terms as actual equality, and expressing $\Gamma \vdash A$ instead as $\Gamma, A \vdash$, led Voevodsky to defining a B-system to consist of families of sets $(B_n, \widetilde{B}_n)_{n\in\mathbb{N}}$, intuitively denoting, for any $n \in \mathbb{N}$, contexts of length n, and terms in a context of length $n-1$ together with their types, respectively. Furthermore, any B-system has weakening and substitution operations on B and $\widetilde{B}$, and a projection map $\widetilde{B}_n \to B_n$ giving, for each term, its context and type. B-systems play a crucial role in Voevodsky's construction of a syntactic C-system from a signature [7].

The main result of this work is the construction of an equivalence of categories between the category **Csys** of C-systems and the category **Bsys** of B-systems. This equivalence is a refinement of an equivalence between more general structures introduced in this work, called CE-systems and E-systems, respectively.

*This work was funded by EPSRC grant EP/T000252/1.
†This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

An **E-system** $\mathbb{E}$ consists of a category $\mathcal{F}$ with a chosen terminal object [] called **empty context**, together with a set $T(A)$ for every arrow A in $\mathcal{F}$, whose domain we write as $\Gamma.A$ where Γ is its codomain, and

1. for every $A \in \mathcal{F}/\Gamma$, a weakening morphism, *i.e.* a functor $W_A \colon \mathcal{F}/\Gamma \to \mathcal{F}/\Gamma.A$ together with an action $T(P) \to T(W_A(P))$ for every $B \in \mathcal{F}/\Gamma$ and $P \in \mathcal{F}/\Gamma.B$,
2. for every $A \in \mathcal{F}/\Gamma$ and $x \in T(A)$, a substitution morphism, *i.e.* a functor $S_x \colon \mathcal{F}/\Gamma.A \to \mathcal{F}/\Gamma$ together with an action $T(P) \to T(S_x(P))$ for every $B \in \mathcal{F}/\Gamma.A$ and $P \in \mathcal{F}/\Gamma.A.B$,
3. for every $A \in \mathcal{F}/\Gamma$, an element $\mathsf{idtm}_A \in T(W_A(A))$

that are required to satisfy a number of equations. In particular, weakening and substitution morphisms are required to distribute over themselves and one over the other. Given two E-systems $\mathbb{E}$ and $\mathbb{D}$, an **E-homomorphism** $F \colon \mathbb{E} \to \mathbb{D}$ consists of a functor $F \colon \mathcal{F}_{\mathbb{E}} \to \mathcal{F}_{\mathbb{D}}$ between the underlying categories strictly preserving the empty context, and an action $T_{\mathbb{E}}(A) \to T_{\mathbb{D}}(FA)$ for every arrow A in $\mathcal{F}_{\mathbb{E}}$, which distribute over weakening and substitution morphisms and preserve the elements idtm.

A **CE-system** $\mathbb{A}$ consists of an identity-on-objects functor $I \colon \mathcal{F} \to \mathcal{C}$, a chosen terminal object $\top$ in $\mathcal{F}$ and, for every $f \colon \Delta \to \Gamma$ in $\mathcal{C}$ and $A \in \mathcal{F}/\Gamma$, a choice of arrows $f^*A \in \mathcal{F}/\Delta$ and $\pi_2(f, A) \colon \Delta.f^*A \to \Gamma.A$ in $\mathcal{C}$ making the obvious square in $\mathcal{C}$ a pullback. This choice is required to be functorial both in A and f. When $\top$ is also terminal in $\mathcal{C}$, we say that $\mathbb{A}$ is **rooted**. CE-systems naturally form a category **CEsys**. We write **rCEsys** for its full subcategory on the rooted CE-systems.

A category $\mathcal{C}$ with a terminal object 1 is said to be **stratified** if it is a rooted tree, more precisely, if there exists a stratification functor $L \colon \mathcal{C} \to (\mathbb{N}, \geq)$, i.e. a Conduché functor with discrete fibres and such that $L(1) = 0$. For example, the category $\mathcal{F}$ generated by the projections of a C-system is stratified. A functor between stratified categories is stratified if it commutes with the stratification functors. A (C)E-system is stratified if the underlying category $\mathcal{F}$ is stratified and weakening and substitution (resp. pullback) are stratified functors. A (C)E-homomorphism is stratified if the underlying functor (resp. the component on $\mathcal{F}$) is so.

Theorem 1. Bsys *is equivalent to the subcategory of* **Esys** *on the stratified E-systems and stratified E-homomorphisms.*

Theorem 2. Csys *is equivalent to the subcategory of* **CEsys** *on the stratified rooted CE-systems and stratified CE-homomorphism.*

We construct a functor E2CE: **Esys** $\to$ **CEsys** and prove the following.

Theorem 3.

1. *The functor* E2CE *is full and faithful and has a right adjoint* CE2E*.*
2. *The adjunction* E2CE $\dashv$ CE2E *restricts to an equivalence between* **Esys** *and* **rCEsys***.*
3. *The equivalence* **Esys** $\simeq$ **rCEsys** *restricts to an equivalence between* **Bsys** *and* **Csys***.*

In the case of B-systems and C-systems, the functor E2CE provides a crucial ingredient in the construction of the initial C-system from a given system of symbols and typing judgements.

Questions still open are to implement this result in a proof assistant, and to extend the equivalence between B-systems and C-systems to the usual type formers.

References

[1] John Cartmell. Generalised algebraic theories and contextual categories. *Ann. Pure Appl. Logic*, 32(0):209 – 243, 1986.

[2] Vladimir Voevodsky. B-systems. *http://arxiv.org/abs/1410.5389*, 2014. Latest version available at https://www.math.ias.edu/Voevodsky/files/files-annotated/Dropbox/Unfinished_papers/Type_systems/Notes_on_Type_Systems/Bsystems/B_systems_current.pdf.

[3] Vladimir Voevodsky. A C-system defined by a universe category. *Theory Appl. Categ.*, 30:Paper No. 37, 1181–1215, 2015.

[4] Vladimir Voevodsky. Products of families of types and (π, λ)-structures on C-systems. *Theory Appl. Categ.*, 31:Paper No. 36, 1044–1094, 2016.

[5] Vladimir Voevodsky. Subsystems and regular quotients of C-systems. In *A panorama of mathematics: pure and applied*, volume 658 of *Contemp. Math.*, pages 127–137. Amer. Math. Soc., Providence, RI, 2016.

[6] Vladimir Voevodsky. C-systems defined by universe categories: presheaves. *Theory Appl. Categ.*, 32:Paper No. 3, 53–112, 2017.

[7] Vladimir Voevodsky. C-system of a module over a Jf-relative monad. February, 2016. Submitted for publication.

Types are Internal ∞-groupoids

Antoine Allioux[1], Eric Finster[2], and Matthieu Sozeau[3]

[1] Inria & Université de Paris, France
antoine.allioux@irif.fr
[2] University of Cambridge, United Kingdom
ericfinster@gmail.com
[3] Inria, France
matthieu.sozeau@inria.fr

Introduction An open problem in Homotopy Type Theory is how to define interesting higher-dimensional algebraic structures which are not truncated. This requires to find an elegant way to handle complex combinatorial phenomena so that an infinite tower of data can be specified in a finite fashion.

In particular, deprived of a strict equality — an equality satisfying the uniqueness of identity proof (UIP) — we can not expect to define strict structures such as operads which could be used to present, in turn, weak structures as it is classically done in mathematics.

An attempt to remedy this situation has been to reintroduce an equality satisfying the UIP in systems known as two-level type theories but this separates the type theory into two layers and gives up the homotopy interpretation of all types [2].

We pursue another approach [1] consisting in extending type theory with a certain class of strict structures: polynomial monads. From these structures we are then able to encode a wide range of weak and higher-dimensional structures through a coinductive construction related to the Baez-Dolan construction [3, 5]. This system has been prototyped by extending Agda's type-theory with rewrite rules [4].

Polynomial monads We postulate a universe of polynomial monads $\mathbb{M} : \mathcal{U}$ whose elements are codes for our monads. The data of the functorial part is given by a set of decoding functions (Figure 1). They can be seen as a way to describe the signature of an algebraic theory: the elements of $\mathsf{Idx}\, M$, which we refer to as *indices* serve as the sorts of the theory, and for $i : \mathsf{Idx}\, M$, the type $\mathsf{Cns}\, M\, i$ is the collection of operation symbols whose "output" sort is i. The type $\mathsf{Pos}\, M\, c$ then assigns to each operation a collection of "input positions" which are themselves assigned an index via the function Typ.

$$
\begin{aligned}
&\mathsf{Idx} : \mathbb{M} \to \mathcal{U} \\
&\mathsf{Cns} : (M : \mathbb{M}) \to \mathsf{Idx}\, M \to \mathcal{U} \\
&\mathsf{Pos} : (M : \mathbb{M})\ \{i : \mathsf{Idx}\, M\} \to \mathsf{Cns}\, M\, i \to \mathcal{U} \\
&\mathsf{Typ} : (M : \mathbb{M})\ \{i : \mathsf{Idx}\, M\}\ (c : \mathsf{Cns}\, M i) \\
&\qquad \to \mathsf{Pos}\, M\, c \to \mathsf{Idx}\, M
\end{aligned}
$$

Figure 1: Decoding functions

Next, we equip this data with a unit η and a multiplication μ satisfying some laws endowing the functor with a structure of monad (Figure 2). μ sends a well-typed depth-2 tree of constructors of M to a constructor of M while preserving the positions. Crucially, it is subject to rewrite rules making it definitionally associative and unital with unit η.

$$
\begin{aligned}
&\eta : (M : \mathbb{M})\ (i : \mathsf{Idx}\, M) \to \mathsf{Cns}\, M\, i \\
&\mu : (M : \mathbb{M})\ \{i : \mathsf{Idx}\, M\}\ (c : \mathsf{Cns}\, M\, i) \\
&\quad \to (\delta : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\, M\, (\mathsf{Typ}\, M\, c\, p)) \\
&\quad \to \mathsf{Cns}\, M\, i
\end{aligned}
$$

Figure 2: Operations of the monad

For every monad we want to introduce, we first postulate its code $M : \mathbb{M}$ then we extend the decoding functions with the relevant data. This is accomplished using rewrite rules.

Pullback monad Given a monad M and a type family $X_0 : \mathsf{Idx}\, M \to \mathcal{U}$, we define the pullback monad $\mathsf{Pb}\, M\, X_0$ which reindexes constructors of M. Its type of indices is $\sum_{i:\mathsf{Idx}\, M} X_0\, i$ and constructors indexed by (i, x) are constructors $c : \mathsf{Cns}\, M\ i$ together with a decoration $\nu : (p : \mathsf{Pos}\, M\, c) \to X_0\ (\mathsf{Typ}\, M\, c\, p)$. The index at position p is then given by $(\mathsf{Typ}\, M\, c\, p, \nu\, p)$. The monadic structure is obvious: the multiplication uses the multiplication of the underlying monad M and forgets the decoration of the inner edges. As for the unit, it uses the unit of M and decorates its input with the index of the output.

Slice monad Given a monad M, we define the monad $\mathsf{Slice}\, M$ whose indices are $\sum_{i:\mathsf{Idx}\, M} \mathsf{Cns}\, M\, i$ and whose constructors β indexed by (i, c) are well-formed trees of constructors of M which multiply to c by μ. The positions of β correspond to the nodes of the tree. The multiplication μ substitutes trees for the nodes of the tree corresponding to the root node. The unit promotes a constructor of M to a corolla — a tree with a single node.

Weak algebraic structures Now, consider the type $\mathsf{Idx}\,(\mathsf{Slice}\,(\mathsf{Pb}\, M\, X_0))$. Its elements are of the form $(i\,, x\,, c\,, \nu)$ where $c : \mathsf{Cns}\, M\ i$ is a constructor whose inputs are decorated with $\nu : (p : \mathsf{Pos}\, M\, c) \to X_0\ (\mathsf{Typ}\, M\, c\, p)$ and whose output is decorated with $x : X_0\ i$. Therefore, we can understand a type family $X_1 : \mathsf{Idx}\,(\mathsf{Slice}\,(\mathsf{Pb}\, M\, X_0)) \to \mathcal{U}$ as a relation over constructors of M decorated with elements in X_0. We say that the family X_1 is *multiplicative* if for all constructors c and for all decorations of inputs ν it is the case that the type $\sum_{x:X_0\, i} X_1\,(i\,, x\,, c\,, \nu)$ is contractible. This defines a multiplication whose result is the first component of this sigma type.

Opetopic Types Notice that X_1 is of type $\mathsf{Idx}\, M_1 \to \mathcal{U}$ for some monad M_1 built from M and X_0. We can iterate this process and consider a family $X_2 : \mathsf{Idx}\,(\mathsf{Slice}\,(\mathsf{Pb}\, M_1\ X_1)) \to \mathcal{U}$. It can be seen that, if both X_1 and X_2 are multiplicative, the multiplication induced by the multiplicative structure on X_1 is associative and unital up to a propositional identity.

```
record OpetopicType (M : 𝕄) : 𝒰₁ where
  𝒞 : Idx M → 𝒰
  ℛ : OpetopicType (Slice (Pb M 𝒞))
```

Figure 3: Opetopic type

Actually, we can collect these families in a coinductive fashion: this is our definition of opetopic type (Figure 3). It is to be seen as an infinite collection of well-typed n-cells. We say that such an opetopic type is fibrant if the families X_n $(n > 0)$ are multiplicative. In that case, pasting diagrams of n-cells $(n > 0)$ can be composed and this composition is witnessed by a $(n+1)$-cell. Moreover, the composition is associative and unital up to a propositional identity.

All is set to introduce our definition of ∞-groupoids: $\infty\text{-}\mathsf{Grp} = \sum_{(X:\mathsf{OpetopicType}\,\mathsf{Id})} \mathsf{is\text{-}fibrant}\, X$. Here Id is the identity monad which has one sort and one constructor with a single input.

Finally, we can state our main result: there is a canonical equivalence

$$\mathcal{U} \simeq \infty\text{-}\mathsf{Grp}$$

In other words, every type admits the structure of an ∞-groupoid in our sense, and that structure is unique.

References

[1] Antoine Allioux, Eric Finster, and Matthieu Sozeau. Types are internal ∞-groupoids. *CoRR*, abs/2105.00024, 2021.

[2] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *CoRR*, abs/1705.03307, 2017.

[3] John C Baez and James Dolan. Higher-dimensional algebra iii. n-categories and the algebra of opetopes. *Advances in Mathematics*, 135(2):145–206, 1998.

[4] Jesper Cockx. Type theory unchained: Extending agda with user-defined rewrite rules. In *25th International Conference on Types for Proofs and Programs (TYPES 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[5] Joachim Kock, André Joyal, Michael Batanin, and Jean-François Mascari. Polynomial functors and opetopes. *Advances in Mathematics*, 224(6):2690–2737, 2010.

A container model of type theory*

Thorsten Altenkirch[1] and Ambrus Kaposi[12]

[1] University of Nottingham, Nottingham, United Kingdom
[2] Eötvös Loránd University, Budapest, Hungary

Introduction

We present a model of type theory where types are interpreted as containers [1] aka polynomial functors. The motivation for this construction is to provide a container semantics for inductive-inductive types and quotient inductive-inductive types.

Consider the core of our usual example of an inductive-inductive type:

```
data Con : Set
data Ty : Con → Set
_,_ : (Γ : Con) → Ty Γ → Con
Π : (Γ : Con) (A : Ty Γ) (B : Ty (Γ , A)) → Ty Γ
```

Since the constructor Π uses the previously defined constructor _,_ in its domain there is no hope of a functorial semantics where an inductive type is the initial algebra of a container. Instead we have to interpret the domain of a constructor as a functor L from the category of algebras induced by the previous constructors to Set, and the codomain a functor from the category of elements of L to Set. The semantics of a constructor with regard to a fixed algebra X is given by (x : L X) → R (X , x). This is explained in detail in [2] where L is an arbitrary functor and types are interpreted using the usual presheaf semantics of type theory.

Such a functorial semantics is too generous because there are many functors which do not have initial algebras. Hence we want to use containers to model strict positivity semantically. As a first step we show that containers do form a model of basic type theory, i.e. a category with families.

Tamara von Glehn also presents a model of type theory using polynomial functors [7] using comprehension categories as notion of model. The same model was presented by Atkey [4] and by Kovács [6] using categories with families (CwFs). This model has the same contexts and substitutions as ours but different types and terms (see below).

In this abstract when we write Set we mean Agda's universe of types and we assume uniqueness of identity proofs (see also the Discussion).

The model

A container (or polynomial functor) is given by a set of shapes S : Set and a family of positions P : S → Set. This gives rise to a functor S ◁ P : Set → Set which on objects is given by (S ◁ P) X = Σ s : S . P s → X. Given containers S ◁ P and T ◁ Q a morphism is given by a function on shapes f : S → T and a family of functions on positions g : (s : S) → Q (f s) → P s — note the change of direction. This gives rise to a natural transformation f ◁ g : (X : Set) → (S ◁ P) X → (T ◁ Q) X given by

*Supported by USAF, Airforce office for scientific research, award FA9550-16-1-0029.

$(f \lhd g)\ X\ (s , p) = (f\ s, \lambda\ s \to p \circ g\ s)$. Using the Yoneda lemma we can show that every natural transformation between containers arises this way (i.e. the evaluation functor from the category of containers to the functor category is full and faithful).

We can generalize set-containers to containers over an arbitrary category $\mathbb{C}$, i.e. covariant functors $\mathbb{C} \to \mathsf{Set}$ using $S : \mathsf{Set}$ and $P : S \to \mathbb{C}$ and $(S \lhd P)\ X = \Sigma\ s : S\ .\ \mathbb{C}\ (P\ s, X)$ and morphisms are given by $f : S \to T$ and $g : (s : S) \to \mathbb{C}\ (Q\ (f\ s) , P\ s)$ with the same definition of the natural transformation.

We define a category with families (CwF) which are the algebras of an intrinsic presentation of Type Theory as given in [3]. The objects corresponding to contexts are set containers and the morphisms are container morphisms. We write Con for this category. Below we sketch some aspects of the construction, for details please check our (incomplete) Agda formalisation [5].

To interpret types we define a functor $\mathsf{Ty} : \mathsf{Con} \to \mathsf{Set}_1$ on objects: given $\Gamma : \mathsf{Con}$, an $A : \mathsf{Ty}\ \Gamma$ is given by a container $A : \int \Gamma \to \mathsf{Set}$. Here $\int \Gamma$ is the category of elements of Γ with objects $\Sigma\ X : \mathsf{Set}\ .\ \Gamma\ X$ and morphisms $\int \Gamma\ ((X , x) , (Y , y)$ are given by a function $f : X \to Y$ such that $\Gamma\ f\ x = y$.

In contrast, a type in von Glehn's model [7] over a context $\Gamma = S \lhd P$ is a container $A : \int (\mathsf{Const}\ S) \to \mathsf{Set}$ where $\mathsf{Const}\ S$ is the constant S presheaf. Hence types there are dependent only on shapes, but not positions.

The interpretation of terms is given by $\mathsf{Tm} : \int \mathsf{Ty} \to \mathsf{Set}$. On an object of $\int \mathsf{Ty}$, i.e. a $\Gamma : \mathsf{Con}$ and $A : \int \Gamma \to \mathsf{Set}$ a term is given by a *dependent natural transformation* $(X : \mathsf{Set})\ (x : \Gamma\ X) \to A\ (X , x)$. Assuming that $\Gamma = S_\Gamma \lhd P_\Gamma$ and $A = S_A \lhd P_A$ using the *dependent Yoneda lemma* we can show that this corresponds to $f : S_\Gamma \to S_A$ and $g : (s : S_A) \to \int \Gamma\ (P_A\ s , (P_\Gamma\ (P^s_A\ s) , P^s_A\ s , \mathsf{id}))$. This can be further simplified using dependent types — see our Agda code.

Given $A : \mathsf{Ty}\ \Gamma$ we write $P^X_A : S_A \to \mathsf{Set}$, $P^s_A : S_A \to S_\Gamma$ and $P^g_A : (s : S_A) \to P_\Gamma\ (P^s_A\ s) \to P^X_A\ s$ for the projections of $P_A : S_A \to \int \Gamma$.

The empty context is given by the terminal object in Con which is $1 \lhd 0$. Assuming a $\Gamma : \mathsf{Con}$ and $A : \mathsf{Ty}\ \Gamma$ we construct the context extension Γ , A as $S_A \lhd P^X_A$. We can verify the universal property — see the Agda code.

The definition of type substitution requires pushouts which can be defined using a quotient inductive type (QIT). That is given $f \lhd g : \mathsf{Con}\ (\Delta , \Gamma)$ and $A : \mathsf{Ty}\ \Gamma$ we construct $A[f \lhd g] = S \lhd P : \mathsf{Ty}\ \Gamma$. We obtain S as the pullback of f and P^s_A. Given $s : S , P\ s$ is the pushout of $g\ s$ and $P^g_A\ s$ (this only type-checks after transporting along the equations).

Discussion

For our application we need to construct the model wrt to an already constructed category of algebras instead of Set. Hence we need to verify that pushouts exists. We need a constructive variant of locally presentable categories here, which have the required colimits.

For our application we only need a basic CwF structure but we can also interpret Σ-types and we expect that we can interpret Π-types in our model, the latter giving rise to higher-order abstract syntax.

One issue with our construction is that types and contexts are not h-sets hence we need to address the coherence issues. We believe that this fits very well with a generalisation of CwFs where types can be groupoids (or 1-types) which we are also investigating (coherent CwFs).

References

[1] Michael Gordon Abbott. *Categories of containers*. PhD thesis, University of Leicester, 2003.

[2] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 293–310. Springer, Cham, 2018.

[3] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 18–29, New York, NY, USA, 2016. ACM.

[4] Robert Atkey. Interpreting dependent types with containers. Talk given at the MSP101 seminar of the University of Strathclyde. Slides available at https://bentnib.org/docs/tt-in-containers.pdf and formalisation at https://gist.github.com/bobatkey/0d1f04057939905d35699f1b1c323736., June 2020.

[5] Ambrus Kaposi and Thorsten Altenkirch. Agda formalisation of the container model of type theory, available at https://bitbucket.org/akaposi/qiitcont, 2021.

[6] András Kovács. The container model of type theory. Talk given at the type theory seminar of Eötvös Loránd University. Formalisation available at https://bitbucket.org/akaposi/tipuselmelet/src/master/notes/seminar20200623_container.agda., June 2020.

[7] Tamara von Glehn. *Polynomials and models of type theory*. PhD thesis, University of Cambridge, 2014.

A Gradual Intersection Typed Calculus

Pedro Ângelo[1,2] and Mário Florido[1,3]

[1] Faculdade de Ciências & LIACC, Universidade do Porto
[2] up201208761@up.pt
[3] amflorid@fc.up.pt

1 Introduction

Types have been broadly used to verify program properties and reduce or, in some cases, eliminate run-time errors. Programming languages adopt either static typing or dynamic typing to prevent programs from erroneous behavior. Static typing is useful for compile time detection of type errors. Dynamic typing is done at run-time and enables rapid software development. Integration of static and dynamic typing has been a quite active subject of research in the last years under the name of *gradual typing* [9, 10, 18, 19, 28–30].

Intersection types, introduced by [11] in 1980, give a type theoretical characterization of strong normalization. Several other contributions followed, making intersection types a rich area of study [3, 4, 7, 12, 16, 22, 23], also used in practice in programming language design and implementation [5, 8, 14, 17, 26, 31]. Although the type inference problem for intersection types is not decidable in general, it becomes decidable for finite rank fragments of the general system [22]. Rank 2 intersection types [2, 16, 20, 21] are particularly interesting because they type more terms than the Hindley-Milner type system [15, 25], while maintaining the same complexity of the typability problem.

In this paper, we present a gradually typed calculus with rank 2 intersection types. To gradually shift type checking to run-time, one needs to annotate lambda-abstractions with the dynamic type, Dyn, which matches with any type. Therefore, gradual type systems have an intrinsic need for explicit type annotations.

2 Annotated Terms

Intersection types were originally designed as descriptive type assignment systems à la Curry, where types are assigned to untyped terms. Prescriptive versions of intersection type systems, supporting typed terms with type annotations in the λ-abstractions, are not trivial [6, 16, 24, 26, 27, 32]. We faced similar problems in our typed calculus to add dynamic type annotations to individual occurrences of formal parameters. As an example, consider the following annotated λ-expression, where we need to instantiate σ in order to make the expression well-typed: $(\lambda x^{Dyn \wedge (Int \to Int)} \,.\, x\ x)\ (\lambda y^{\sigma} \,.\, y)$. This expression can be typed with Dyn, because $\lambda x^{Dyn \wedge (Int \to Int)} \,.\, x\ x$ has type $Dyn \wedge (Int \to Int) \to Dyn$ and $\lambda y^{\sigma} \,.\, y$ may have two types: $(Int \to Int) \to Int \to Int$, with σ equal to $Int \to Int$, and $Int \to Int$, with σ equal to Int. The question now is how to choose the right type for σ. One might be tempted to write the term as $\lambda y^{(Int \to Int) \wedge Int} \,.\, y$, however that would result in the expression being typed as either $(Int \to Int) \wedge Int \to Int \to Int$ or $(Int \to Int) \wedge Int \to Int$, both of which are incorrect. Several solutions have been presented to this problem [6, 24, 26, 27, 32]. Intersection-types à la Church [24] tackled this challenge by dividing the calculus into two: marked-terms encode λ-calculus terms and connect to proof-terms via a variable mark, while the latter carries logical information in the form of proof trees with type annotations. Although technically sound and

clean, the added overhead of carrying two distinct terms, as well as the indirection arising from the connection between them, is too heavy for our specific purpose, since integrating with gradual typing will already mean adding a significant level of extra complexity. Branching Types [32] encoded different derivations directly into types, by assigning to types a kind that keeps track of the shapes of each derivation. Although being an elegant way of dealing with explicit annotations we found more recent approaches to be more viable and useful for integration with gradual typing. Another typed language with intersection types is Forsythe [26], however it was not considered simply because some terms in this system lack correct typings when fully annotated, e.g. there is no annotated version of $(\lambda x.(\lambda y.x))$ with type $(\tau \to \tau \to \tau) \wedge (\rho \to \rho \to \rho)$. A Typed Lambda Calculus with Intersection Types [6] introduces parallel terms, where each component is annotated, resulting in the typing of the parallel term with an intersection type. Besides allowing type annotations, parallel terms also make easier the definition of dynamic type checking of terms typed by an intersection type. Thus, due mainly to this simplicity and elegant design, we chose to use [6] as the basis upon which we built our system. Our gradual type system makes use of parallel terms of the form $M_1 \mid \ldots \mid M_n$, where each M_i, for $i \in 1..n$, is a term with a unique type assigned to it. In the example above, the expression would now be annotated as $(\lambda x^{Dyn \wedge (Int \to Int)} \,.\, x\ x)\ (\lambda y^{Int \to Int} \,.\, y \mid \lambda z^{Int} \,.\, z)$, where the type of the argument is $((Int \to Int) \to Int \to Int) \wedge (Int \to Int)$.

3 Contributions

Standard gradual types enable to declare every occurrence of formal function parameters as dynamically typed. Our system, using intersection types, enables to declare some occurrences of a formal parameter as dynamically typed and other occurrences as statically typed. This gives a new fine-grained definition of dynamicity which is only possible by the use of intersection types. Thus, the main contributions of our paper are:

1. a gradual intersection typed calculus with rank 2 intersection types. In this calculus, formal parameters in abstractions are annotated with gradual intersection types and distinct typed versions of function arguments are written as a parallel term;

2. a compilation procedure into a new cast calculus, which inserts run-time checks to ensure that plausibly correct code is verified at run-time;

3. a type safe operational semantics, where progress and preservation hold, which reduces cast calculus expressions;

4. we show that our calculus has the usual correctness criteria properties for gradual typing [30], mainly that the gradual guarantee holds.

Although originally defined in a programming language context, the logical meaning of the dynamic type is an interesting question, even more relevant in the context of intersection type systems, due to the apparent similarities between the dynamic type and the ω type [13]. Our work can be viewed as a first step towards a proof-theoretical characterization of the dynamic type in the context of intersection types. In our paper, we restrict gradual intersection types to rank 2, for which there is a complete type inference algorithm [1]. We believe that it is possible to adapt the algorithm in [1] to output the whole syntactic tree of annotated parallel terms, given a partially annotated lambda term as input. This would also enable the use of our calculus as an intermediate code in a gradually typed programming language, avoiding the extra effort of programmers to write several annotated copies of function arguments.

References

[1] Pedro Ângelo and Mário Florido. Type inference for rank 2 gradual intersection types. In William J. Bowman and Ronald Garcia, editors, *Trends in Functional Programming (TFP 2019)*, LNCS, pages 84–120. Springer, 2020.

[2] Steffen van Bakel. Rank 2 intersection type assignment in term rewriting. *Fundam. Inf.*, 26(2):141–166, May 1996.

[3] Stephanus Johannes van Bakel. *Intersection type disciplines in lambda calculus and applicative term rewriting systems*. Amsterdam: Mathematisch Centrum, 1993.

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

[5] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & Lambda: a Featherweight Story. *Logical Methods in Computer Science*, Volume 14, Issue 3, September 2018.

[6] Viviana Bono, Betti Venneri, and Lorenzo Bettini. A typed lambda calculus with intersection types. *Theor. Comput. Sci.*, 398(1–3):95–113, May 2008.

[7] Sébastien Carlier and J.B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. *Electronic Notes in Theoretical Computer Science*, 136:173 – 202, 2005. Proceedings of the Third International Workshop on Intersection Types and Related Systems (ITRS 2004).

[8] Avik Chaudhuri. Flow: Abstract interpretation of javascript for type checking and beyond. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, page 1, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Matteo Cimini and Jeremy G. Siek. The gradualizer: A methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 443–455, New York, NY, USA, 2016. ACM.

[10] Matteo Cimini and Jeremy G. Siek. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 789–803, New York, NY, USA, 2017. ACM.

[11] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 10 1980.

[12] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

[13] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. Functional characterization of some semantic equalities inside lambda-calculus. In *Automata, Languages and Programming, 6th Colloquium, July 16-20, 1979*, volume 71 of *Lecture Notes in Computer Science*, pages 133–146. Springer, 1979.

[14] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In Rastislav Královič and Paweł Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006*, pages 1–23, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[15] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[16] Ferruccio Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. Program. Lang. Syst.*, 25(4):401–451, July 2003.

[17] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. *Intersection Types in Java: Back to the Future*, pages 68–86. Springer International Publishing, Cham, 2019.

[18] Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

POPL '15, pages 303–315, New York, NY, USA, 2015. ACM.

[19] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM.

[20] T. Jim. Rank 2 type systems and recursive definitions. Technical report, Cambridge, MA, USA, 1995.

[21] Trevor Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 42–53, New York, NY, USA, 1996. ACM.

[22] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 161–174, New York, NY, USA, 1999. ACM.

[23] A.J. Kfoury and J.B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1):1 – 70, 2004.

[24] Luigi Liquori and Simona Ronchi Della Rocca. Intersection-types à la church. *Information and Computation*, 205(9):1371 – 1386, 2007.

[25] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[26] John C. Reynolds. *Design of the Programming Language Forsythe*, pages 173–233. Birkhäuser Boston, Boston, MA, 1997.

[27] Simona [Ronchi Della Rocca]. Intersection typed λ-calculus. *Electronic Notes in Theoretical Computer Science*, 70(1):163 – 181, 2003. ITRS '02, Intersection Types and Related Systems (FLoC Satellite Event).

[28] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[29] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 7:1–7:12, New York, NY, USA, 2008. ACM.

[30] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[31] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 310–325, New York, NY, USA, 2016. Association for Computing Machinery.

[32] Joe B. Wells and Christian Haack. Branching types. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, pages 115–132, London, UK, UK, 2002. Springer-Verlag.

M-types and Bisimulation

Henning Basold[1] and Daniël Otten[2]

[1] Leiden University, Leiden, The Netherlands
h.basold@liacs.leidenuniv.nl
[2] University of Amsterdam, Amsterdam, The Netherlands
daniel@otten.co

1 Introduction

We often use the intuition that inductive types correspond to initial algebras in the category-theoretical sense, while coinductive types correspond to final coalgebras. However, once we want to make this intuition precise, we face some problems. For inductive types, an induction principle, also known as dependent iteration, can be combined with extensionality to show that inductive types are initial algebras. Trying the same idea naively for coinductive types however makes equality undecidable [2].

In fact, the situation for inductive types is already quite a bit more subtle. Firstly, there is a choice between assuming that the induction principle satisfies its equalities definitionally or only propositionally [7, section 5.5]. Secondly, there are multiple ways to define the induction principle. Fortunately, with function extensionality these turn out to be equivalent [1, 7].

In the case of coinductive types, we are not that fortunate because the standard definition of the induction principle uses dependent functions and can therefore not easily be dualised. Interestingly, the problem of finding a good definition of the coinduction principle is not unique to type theory and already comes up in the theory of coalgebras [6]. In this work, we study different versions of the coinduction principle in Martin-Löf type theory and show under which conditions these are equivalent. The various definitions we consider have different desirable properties (elegance; ease of use; applicability; constructabilty; compositionality).

2 Coalgebras and M-Types

We restrict our attention to strictly positive coinductive types, as these arise from coalgebras for polynomial functors. This avoids complications that more general classes of functors may have within the context of type theory. The polynomial functor for $A\colon \mathrm{Type}$ and $B\colon A \to \mathrm{Type}$ [3] is the functor $P\colon \mathrm{Type} \to \mathrm{Type}$ given by

$$P\,X \coloneqq \textstyle\sum_{(a\colon A)} B\,a \to X,$$
$$P\;f \coloneqq \lambda(a, d).\,(a, f \circ d).$$

A P-coalgebra consists of an object X and a morphism $\mathrm{obs}_X\colon X \to P\,X$. Together with an appropriate notion of morphisms, P-coalgebras form a category. Our interest lies then in finding and studying the final objects in this category.

Intuitively, a P-coalgebra $\mathrm{obs}_X\colon X \to P\,X$ sends a term $x\colon X$ to a value $a\colon A$ and for every $b\colon B\,a$ a new term of X. By iterating obs_X, we can generate a tree of potentially infinite depth. Its nodes are labeled with values $a\colon A$ and have precisely one child for every $b\colon B\,a$. We assume the existence of a type M of trees together with a map $\mathrm{obs}_{\mathsf{M}}\colon \mathsf{M} \to P\,\mathsf{M}$. We now wish to compare different methods of enforcing that such an M-type is the final P-coalgebra, that is, we compare various ways of implementing the coinduction principle in type theories.

We start by defining two kinds of bisimulations, for this we use two different ways to define relations on a type. The first definition comes from category theory and is based on spans:

$\text{SpanRel}\ X \coloneqq \sum_{(R\colon \text{Type})} (R \to X)^2,$
$\text{SpanRelMor}\ (R, \rho_0, \rho_1)\ (S, \sigma_0, \sigma_1) \coloneqq \sum_{(f\colon R \to S)} ((\sigma_0 \circ f \equiv \rho_0) \times (\sigma_1 \circ f \equiv \rho_1)).$

This leads to a type of bisimulation also known as AM-bisimulation [6]:

$\text{SpanBisim}\ (X, \text{obs}_X) \coloneqq \sum_{(R\colon \text{Coalg})} (\text{CoalgMor}\ R\ (X, \text{obs}_X))^2,$
$\text{SpanBisimMor}\ (R, \rho_0, \rho_1)\ (S, \sigma_0, \sigma_1) \coloneqq \sum_{(f\colon \text{CoalgMor}\ R\ S)} ((\sigma_0 \circ f \equiv \rho_0) \times (\sigma_1 \circ f \equiv \rho_1)).$

The second definition of a relation is more standard for type theory, using a dependent type:

$\text{DepRel}\ X \coloneqq X \to X \to \text{Type},$
$\text{DepRelMor}\ R\ S \coloneqq \prod_{(x_0, x_1\colon X)} (R\ x_0\ x_1 \to S\ x_0\ x_1).$

This leads to a type of bisimulation also known as HJ-bisimulation [6]:

$$\begin{aligned}\text{LiftingBisim}\ (X, \text{obs}_X) \coloneqq{}& \textstyle\sum_{(R\colon \text{DepRel}\ X)} \prod_{(x_0, x_1\colon X)} R\ x_0\ x_1 \to \\ & \textstyle\sum_{(p\colon \text{pr}_0(\text{obs}_X\ x_0) \equiv \text{pr}_0(\text{obs}_X\ x_1))} \prod_{(b_0\colon B\ (\text{pr}_0(\text{obs}_X\ x_0)))} \\ & R\ (\text{pr}_2\ (\text{obs}_X\ x_0)\ b_0)\ (\text{pr}_2\ (\text{obs}_X\ x_1)\ (\text{tra}_B\ p\ b_0)).\end{aligned}$$

3 Notions of Final Coalgebras

It is common to assume that for any P-coalgebra (X, obs_X) we have a dependent function coiter: $\text{CoalgMor}\ (X, \text{obs}_X)\ (\mathsf{M}, \text{obs}_{\mathsf{M}})$. This makes M-types weakly final, as there exists a morphism to it from all other P-coalgebras. However, the uniqueness of this morphism is not guaranteed. We consider three assumptions that are sufficient to obtain uniqueness:

$$\begin{aligned}\text{IsFinM}\ (\mathsf{M}, \text{obs}_{\mathsf{M}}) \coloneqq{}& \textstyle\prod_{((X, \text{obs}_X)\colon \text{Coalg})} \text{IsContr}\ (\text{CoalgMor}\ (X, \text{obs}_X)\ (\mathsf{M}, \text{obs}_{\mathsf{M}})), \\ \text{IsCohM}\ (\mathsf{M}, \text{obs}_{\mathsf{M}}) \coloneqq{}& \textstyle\prod_{((X, \text{obs}_X)\colon \text{Coalg})} \text{CoalgMor}\ (X, \text{obs}_X)\ (\mathsf{M}, \text{obs}_{\mathsf{M}}) \times \\ & \textstyle\prod_{((f_0, \text{com}_{f_0}), (f_1, \text{com}_{f_1})\colon \text{CoalgMor}\ (X, \text{obs}_X)\ (\mathsf{M}, \text{obs}_{\mathsf{M}}))} \\ & \textstyle\sum_{(p\colon f_0 \equiv f_1)} \prod_{(x\colon X)} \\ & \text{ap}_{(\lambda f.\, \text{obs}_{\mathsf{M}}\ (f\ x))}\ p \cdot \text{apply}^{\equiv}\ \text{com}_{f_1}\ x \equiv \\ & \text{apply}^{\equiv}\ \text{com}_{f_0}\ x \cdot \text{ap}_{(\lambda f.\, P\ f\ (\text{obs}_X\ x))}\ p, \\ \text{IsSpanBisimM}\ (\mathsf{M}, \text{obs}_{\mathsf{M}}) \coloneqq{}& \textstyle\prod_{((X, \text{obs}_X)\colon \text{Coalg})} \text{CoalgMor}\ (X, \text{obs}_X)\ (\mathsf{M}, \text{obs}_{\mathsf{M}}) \times \\ & \textstyle\prod_{((R, \rho_0, \rho_1)\colon \text{SpanBisim}\ (\mathsf{M}, \text{obs}_{\mathsf{M}}))} \\ & \text{IsContr}\ (\text{SpanBisimMor}\ (R, \rho_0, \rho_1)\ ((\mathsf{M}, \text{obs}_{\mathsf{M}}), \text{id}_{\mathsf{M}}, \text{id}_{\mathsf{M}})).\end{aligned}$$

Here the first and second definitions are similar to dual definitions for W-types [7, chapter 5]. The third definition is based on our first notion of bisimulation. We are working on a fourth definition, $\text{IsLiftingBisimM}\ (\mathsf{M}, \text{obs}_{\mathsf{M}})$, based on our second notion of bisimulation. Currently we have a version that is weaker than the other definitions, we want to make it equivalent.

In our work we show the following implications by constructing functions between the types:

$$\text{IsFinM} \rightleftarrows \text{IsSpanBisimM} \underset{\text{funext}}{\rightleftarrows} \text{IsCohM} \longrightarrow \text{IsLiftingBisimM}$$

Here the arrow marked with 'funext' uses the axiom of function extensionality. We are still working on the last missing arrow. These results have largely been formalised in the proof assistant `Agda` [4, 5]. The code can be accessed at https://github.com/DDOtten/M-types.

References

[1] Steve Awodey, Nicola Gambino, and Kristina Sojakova. Inductive types in homotopy type theory. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*, pages 95–104. IEEE, 2012.

[2] Ulrich Berger and Anton Setzer. Undecidability of Equality for Codata Types. In *CMCS 2018, Revised Selected Papers*, pages 34–55, 2018.

[3] Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154(1):153–192, 2013.

[4] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

[5] Programming Language group on Agda. *Agda Documentation*. http://wiki.portal.chalmers.se/agda/, Accessed: 2020.

[6] Sam Staton. Relating coalgebraic notions of bisimulation. *Logical Methods in Computer Science*, Volume 7, Issue 1, March 2011.

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

The syntactic tale of the oracle and the trees
A model of continuity in a dependent setting

Martin Baillon[1] and Pierre-Marie Pédrot[1]

INRIA, France

A folklore result from computability theory is that any computable function must be continuous. There are many ways to prove, or even merely state, this theorem, since it depends in particular on how computable functions are represented. Assuming we pick the λ-calculus as our favorite computational system, one of the most straightforward paths boils down to building a semantic model for it, typically some flavor of *Complete Partial Orders* (CPOs). By construction, CPOs being a specific kind of topological spaces, all functions are then interpreted as continuous functions in the model. For some types simple enough such as $\mathbb{R} \to \mathbb{R}$, CPO-continuity implies continuity in the traditional sense, thus proving the claim.

Instead of going down the semantic route, Escardó developed an alternative syntactic technique called *effectful forcing* [3] to prove the continuity of all functions of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ that are definable in System T. While semantic models such as CPOs are crafted inside a non-computational metatheory, Escardó's technique amounts to building a model of System T inside intensional Martin-Löf type theory (MLTT), that is, in a proper programming language. The *effectful* epithet is justified by the fact it intuitively consists in embedding System T inside two impure extensions, each featuring a different kind of side-effects, and by constraining them via a logical relation. This argument being purely syntactic, it can be leveraged to interpret much richer languages than System T. We describe here how to generalize the latter technique so as to recover his continuity result for a dependent type theory similar to MLTT.

Unfortunately, since Escardó's model introduces observable side-effects, the type theory resulting from our generalization needs to be slightly weakened down, or would otherwise be inconsistent [6]. We thus provide a model of Baclofen type theory (BTT) [7] rather than MLTT. BTT is a type theory with dependent products and a predicative hierarchy of universes. The main difference with MLTT actually lies in the typing rule for dependent elimination, where the predicate needs to be restricted so as to be linear. For instance, when MLTT has a single dependent eliminator for the type of booleans, BTT features two different eliminators, a non-dependent one, $\mathbb{B}\_\mathbf{case}$, and a strict dependent one, $\mathbb{B}\_\mathbf{rect}$. The latter comes with the following typing rule:

$$\frac{\vdash P : \mathbb{B} \to \square \qquad \vdash u_t : P\ \mathtt{true} \qquad \vdash u_f : P\ \mathtt{false}}{\vdash \mathbb{B}\_\mathbf{rect}\ P\ u_t\ u_f : \Pi(b : \mathbb{B}).\, \theta_{\mathbb{B}}\ P\ b}$$

where $\theta_{\mathbb{B}}$ constant is the *storage operator*, getting rid of side effects. We have the following reduction rules:

1. $\theta_{\mathbb{B}}\ P\ \mathtt{true} \equiv P\ \mathtt{true}$
2. $\theta_{\mathbb{B}}\ P\ \mathtt{false} \equiv P\ \mathtt{false}$
3. $\theta_{\mathbb{B}}\ P\ \beta \equiv \mathtt{unit}$ for any β non standard inhabitant of $\mathbb{B}$

Theorem 1. *Any function* $\vdash_{\mathsf{BTT}} f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ *is continuous.*

Following Escardó, the proof goes by considering the operator $\mathfrak{D} : \square \to \square$, which given a type $A : \square$, associates the type of well-founded, $\mathbb{N}$-branching trees, with inner nodes labeled in $\mathbb{N}$ and leaves labeled in A. In Coq, this amounts to the following inductive definition:

$$\texttt{Inductive}\ \mathfrak{D}\ (A : \square) : \square \quad := \quad \eta : A \to \mathfrak{D}\ A \quad | \quad \beta : (\mathbb{N} \to \mathfrak{D}\ A) \to \mathbb{N} \to \mathfrak{D}\ A.$$

These *dialogue trees* can be seen as functions of type $(\mathbb{N} \to \mathbb{N}) \to A$. Intuitively, every inner node is a call to an oracle $\alpha : \mathbb{N} \to \mathbb{N}$, and the answer is the label of the leaf. This interpretation is implemented by a recursively defined dialogue function:

$$\begin{array}{lcl} \partial & : & \Pi\{A : \square\}\,(\alpha : \mathbb{N} \to \mathbb{N})\,(d : \mathfrak{D}\ A).\, A \\ \partial\ \alpha\ (\eta\ x) & := & x \\ \partial\ \alpha\ (\beta\ k\ i) & := & \partial\ \alpha\ (k\ (\alpha\ i)) \end{array}$$

Thanks to the well-foundedness of these dialogue trees, it is relatively straightforward to show that functions defined this way are continuous on $\mathbb{N} \to \mathbb{N}$, equipped with the usual Baire topology. The next step is to prove that every BTT-definable function is extensionally equal to such a dialogue.

The effectful forcing technique consists in embedding System T inside two impure extensions, and to translate the latter into MLTT, thus building a weak form of syntactic model [5, 8]. These extensions are:

1. System $\mathsf{T}\Omega$, which is mainly System T with a formal oracle $\Omega : \iota \to \iota$. Its translation $[.]^\omega$ into MLTT is simply the standard interpretation of System T, parameterized by an oracle $\omega : \mathbb{N} \to \mathbb{N}$. We have: $[\![\iota]\!]^\omega = \mathbb{N}$, $[0]^\omega = 0$, $[\texttt{Succ}]^\omega = \texttt{Succ}$, $[\Omega]^\omega = \omega$, etc.

2. the *dialogue interpretation*, which is again $\mathsf{T}\Omega$, but this time translated into MLTT using a non-standard translation $[.]_{\mathfrak{D}}$. Here, $[\![\iota]\!]_{\mathfrak{D}} = \mathfrak{D}\ \mathbb{N}$, $[0]_{\mathfrak{D}} = \eta\ 0$ and every function is interpreted via the bind function of the $\mathfrak{D}$ monad. For instance, $[\texttt{Succ}]_{\mathfrak{D}}$ is the function that applies $\texttt{Succ}$ to every leaf of a tree. Finally, $[\Omega]_{\mathfrak{D}} = \gamma$ where $\gamma : \mathfrak{D}\ \mathbb{N} \to \mathfrak{D}\ \mathbb{N}$ is a well chosen term from MLTT.

A logical relation constrains these two extensions so as to ensure a fundamental property, namely that for every term $t : \iota$ of System T, and for every $\omega : \mathbb{N} \to \mathbb{N}$, we have:

$$[t]^\omega = \partial\ \omega\ [t]_{\mathfrak{D}}.$$

Crucially, $[t]^\omega$ depends on ω whereas $[t]_{\mathfrak{D}}$ does not. From this property and a bit of work, it follows that every function $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ definable in System T is continuous.

In our model, we follow a similar route, albeit for some slight adjustments: we first define the *axiom translation* $[.]_a^\alpha$ from BTT to MLTT, parameterized by an oracle $\alpha : \mathbb{N} \to \mathbb{N}$, that adds α to every context, thus mimicking the first translation, to System $\mathsf{T}\Omega$.

We then define the *branching translation* $[.]_b$ which resembles a lot the Dialogue interpretation but for a crucial change: a type A is not interpreted as $\mathfrak{D}\ A$ but as what would amount to an algebra of the free $\mathfrak{D}$ monad if we assumed `funext`. This change ensures that the *branching translation* provides a model of BTT.

Finally, the logical relation becomes an additional layer of parametricity. More precisely, it is a case of *cheap binary parametricity*, following the taxonomy from Boulier [2]. However, in order to interpret universes we must also require our parametricity to be algebraic with respect to the $\mathfrak{D}$ operator, making it a layer of *algebraic binary parametricity*.

We have formalized in Coq our proof that Escardó's result extends to BTT [1]. We also discuss whether this extension is the best we can hope for, as a naive phrasing of this result would make MLTT inconsistent [4].

References

[1] Martin Baillon. Formalization: Syntactic model of continuity. URL: https://gitlab.inria.fr/mbaillon/city-of-streams.

[2] Simon Boulier. Extending type theory with syntactic models. *Logic in Computer Science [cs.LO]. École nationale supérieure Mines-Télécom Atlantique*, 2018, https://tel.archives-ouvertes.fr/tel-02007839.

[3] Martin Hötzel Escardó. Continuity of Gödel's system T definable functionals via effectful forcing. *Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2013, New Orleans, LA, USA, June 23-25, 2013*, https://doi.org/10.1016/j.entcs.2013.09.010. doi:10.1016/j.entcs.2013.09.010.

[4] Martin Hötzel Escardó and Chuangjie Xu. The inconsistency of a brouwerian continuity principle with the curry-howard interpretation. *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, https://doi.org/10.4230/LIPIcs.TLCA.2015.153. doi:10.4230/LIPIcs.TLCA.2015.153.

[5] Martin Hofmann. *Extensional constructs in intensional type theory.* CPHC/BCS distinguished dissertations. Springer, 1997.

[6] Pierre-Marie Pédrot and Nicolas Tabareau. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.*, 4(POPL):58:1–58:28, 2020. doi:10.1145/3371126.

[7] Pierre-Marie Pédrot and Nicolas Tabareau. An effectful way to eliminate addiction to dependence. *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, 2017, https://doi.org/10.1109/LICS.2017.8005113. doi:10.1109/LICS.2017.8005113.

[8] Pierre-Marie Pédrot Simon Boulier and Nicolas Tabareau. The next 700 syntactical models of type theory. *Certified Programs and Proofs (CPP 2017), Jan 2017, Paris, France. pp.182 - 194*, https://doi.org/10.1016/j.entcs.2013.09.010. doi:10.1145/3018610.3018620.

Equality checking for dependent type theories*

Andrej Bauer and Anja Petković

University of Ljubljana, Ljubljana, Slovenia

Equality checking algorithms are essential components of proof assistants based on type theories [9, 3, 10, 13, 12, 1]. They free the user from the burden of proving equalities, and provide computation-by-normalization engines. Some systems [11, 8, 7] also allow user extensions to the built-in equality checkers, possibly sacrificing completeness and sometimes even soundness. The situation is even more challenging in a proof assistant that supports arbitrary user-definable type theories, such as Andromeda 2 [4, 5], where in general no equality checking algorithm may be available. Still, the proof assistant should provide convenient support for equality checking that works well in the common, well-behaved cases.

We developed an extensible equality checking algorithm and proved it to be sound [6] for a large class of dependent type theories. The algorithm is parameterized by computation rules (β-rules), extensionality rules (inter-derivable with η-rules), and a notion of normal form. It combines and extends algorithms based on type-directed equality checking [14, 2] that intertwine two phases: the type-directed phase applies extensionality rules to reduce the problem to simpler types, while the normalization phase applies computation rules to compute normal forms.

We define precisely what it means for an equality rule to be a computation or an extensionality rule. For this purpose we identify the notion of an object-invertible rule, which guarantees soundness of normalization steps and of type-directed reductions of subsidiary equations. We give simple syntactic criteria for recognizing computation and extensionality rules.

We implemented the algorithm in the Andromeda 2 proof assistant in around 1400 lines of OCaml code. The user needs only provide the equality rules they wish to use, after which the algorithm automatically classifies them either as computation or extensionality rules (and rejects those that are of neither kind), and devises an appropriate notion of normal form. The implementation consults the nucleus to build a trusted certificate of every equality it proves and every term it normalizes. It is easy to experiment with different sets of equality rules and dynamically switch between them. In the case of well-behaved type theories, such as the simply typed lambda calculus or Martin-Löf type theory, the algorithm behaves like well-known standard equality checkers. We do not address completeness and termination, as these depend heavily on the choice of computation and extensionality rules.

Object-invertible, computation and extensionality rules. In an inference rule

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

the *object premises* are those P_i which are type or term judgements, and *equational premises* those that are type or term equations. We say that such a rule is *object-invertible* when the following holds for every instance of it: if the conclusion is derivable (possibly by application of a different rule) then the object premises are derivable.

Object-invertible rules may be used to invert derivable judgements up to equational premises. That is, if a derivable judgement J coincides with some instance of the conclusion C of an object-invertible rule, then we are guaranteed that the corresponding instances of the object premises are also derivable, so only the equational premises must be checked.

*This material is based upon work supported by the U.S. Air Force Office of Scientific Research under award number FA9550-17-1-0326, grant number 12595060, and award number FA9550-21-1-0024. We thank Philipp G. Haselwarter, who was initially contributing to the project, for his support and discussions.

A *type computation rule* is a derivable type equality rule, shown below on the left,

$$\frac{P_1 \quad \cdots \quad P_n}{\vdash A \equiv B} \qquad\qquad \frac{P_1 \quad \cdots \quad P_n}{\vdash A\ \mathsf{type}}$$

such that its left-hand side presupposition, shown above on the right, is object-invertible and deterministic (its conclusion can be instantiated to match a given judgement in at most one way). *Term computation rules* are defined similarly.

An *extensionality rule* is a derivable rule, shown below on the left,

$$\frac{P_1 \quad \cdots \quad P_n \quad \vdash x : C \quad \vdash y : C \quad Q_1 \quad \cdots \quad Q_m}{\vdash x \equiv y : C} \qquad\qquad \frac{P_1 \quad \cdots \quad P_n}{\vdash C\ \mathsf{type}}$$

such that $Q_1, \ldots, Q_m$ are equational premises, and its type presupposition, shown above on the right, is object-invertible and derivable.

Principal arguments and normal forms. A third component of the algorithm is a suitable notion of normal form, which guarantees correct execution of normalization and coherent interaction of both phases of the algorithm. In our setting, normal forms are determined by a selection of *principal arguments*. By varying these, we obtain known notions, such as weak head-normal and strong normal forms (all arguments are declared principal). An expression is said to be in normal form if no computation rule applies to it, and its principal arguments are in normal form.

In the implementation the user may specify the principal arguments directly, or let the algorithm read the principal arguments off the computation rules automatically, as follows: if $\mathsf{s}(u_1, \ldots, u_n)$ appears as a left-hand side of a computation rule, then the principal arguments of s are those u_i's that are *not* metavariables, i.e., matching against them does not automatically succeed, and so they should first be normalized.

Overview of the type-directed equality checking. The equality checking algorithm is parameterized by the underlying type theory, computation rules, extensionality rules, and principal arguments. It has the following mutually recursive parts:

1. *Normalize a type or a term:* normalize the principal arguments, apply a computation rule and recursively check subsidiary equations as they arise; repeat until no computation rule applies.
2. *Check $A \equiv B$:* normalize A and B and structurally compare their normal forms.
3. *Structurally check $A \equiv B$:* compare A and B by an application of a congruence rule, where the principal arguments are recursively compared structurally and the others by the general equality checks.
4. *Check $s \equiv t : A$:*
   (a) *type-directed phase:* normalize A and apply extensionality rules, if any, to reduce the equality to subsidiary equalities,
   (b) *normalization phase:* if no extensionality rules apply, normalize s and t and structurally compare their normal forms.
5. *Structurally check $s \equiv t : A$:* compare s and t by an application of a congruence rule, where the principal arguments are recursively compared structurally and the others by the general equality checks.

References

[1] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proceedings of the ACM on Programming Languages*, 2(POPL), December 2017.

[2] Andreas Abel and Gabriel Scherer. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science*, Volume 8, Issue 1, 2012.

[3] The Agda proof assistant. `https://wiki.portal.chalmers.se/agda/`.

[4] The Andromeda proof assistant. `http://www.andromeda-prover.org/`.

[5] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Christopher A. Stone. Design and implementation of the Andromeda proof assistant. In *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*, volume 97 of *LIPIcs*, pages 5:1–5:31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[6] Andrej Bauer and Anja Petković. An extensible equality checking algorithm for dependent type theories, 2021.

[7] Jesper Cockx. Type theory unchained: Extending Agda with user-defined rewrite rules. In Marc Bezem and Assia Mahboubi, editors, *25th International Conference on Types for Proofs and Programs (TYPES 2019)*, volume 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:27, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[8] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In *22nd International Conference on Types for Proofs and Programs TYPES 2016*, University of Novi Sad, 2016.

[9] The Coq proof assistant. `https://coq.inria.fr/`.

[10] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *25th International Conference on Automated Deduction (CADE 25)*, August 2015.

[11] The Dedukti logical framework. `https://deducteam.github.io`.

[12] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019.

[13] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL), December 2019.

[14] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 7(4):676–722, 2006.

Type theories without contexts*

Andrej Bauer[1] and Philipp G. Haselwarter[2]

[1] University of Ljubljana, Slovenia
[2] Aarhus University, Denmark

We present a general definition of a class of dependent type theories which we call *finitary type theories*. In fact, we provide two variants of such type theories, with and without typing contexts, and show that they are equally expressive by providing translations between them. Our definition broadly follows the development of general dependent type theories [2], encompassing type theories with intuitionistic contexts and type-, term- and the corresponding equality judgements. Examples of theories that can be expressed in this formalism include Martin-Löf type theory with a variety of type formers (e.g. intensional or extensional equality types, universes, inductive types, ...), simply typed λ-calculi; among counterexamples we find theories with linear context, modal type theories, or cubical type theory where the interval has a special role. In contrast to [2], our finitary type theories are specialized to serve as a formalism for implementation of a proof assistant. Indeed, the present work is the theoretical foundation of the Andromeda 2 proof assistant [1], in which type theories are entirely defined by the user. Nevertheless, we expect the notion of finitary type theories to be applicable in other situations and without reference to any particular implementation.

LCF-style proof assistants, such as Andromeda 2, are based on forward-style reasoning in which previously derived judgements are combined, using the constructors of an abstract datatype of judgements, to form new judgements. A question arises how to combine the underlying contexts. For example, given $\Gamma \vdash a : A$ and $\Delta \vdash b : B$, how should Γ and Δ be combined to give a context for $(a, b) : A \times B$?

In dependent type theories the problem of combining contexts is exacerbated by dependencies. Following traditional presentations of logic and of context-free pure type systems Γ_∞ by Geuvers et al. [3], we address the problem by removing the context altogether, and formulate *context-free type theories* in which judgements have no explicit typing contexts. Instead every variable and metavariable occurrence is annotated with its type or meta-level type, respectively.

Consequently, in context-free type theories judgements of the form "$\Theta; \Gamma \vdash \mathcal{J}$" are replaced with just "$\mathcal{J}$" and rather than having $\mathsf{a} : A$ in the context, each occurrence of a is annotated with its type as a^A.

We have to overcome several technical complications, the most challenging of which is the lack of strengthening, which is the principle stating that if $\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash \mathcal{J}$ is derivable and a does not appear in Δ and $\mathcal{J}$, then $\Theta; \Gamma, \Delta \vdash \mathcal{J}$ is derivable. An example of a rule that breaks strengthening is the equality reflection rule familiar from extensional type theory:

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{p} : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}}$$

Because the conclusion elides the metavariable p, it will not record the fact that a variable may have been used in the derivation of the fourth premise. Consequently, we cannot tell what variables ought to occur in the context just by looking at the judgement. As it turns out, variables elided by derivations of equations are the only culprit, and strengthening can be recovered by modifying equality judgements so that they carry additional information about usage of variables in the form of *assumption sets*.

*This material is based upon work supported by the U.S. Air Force Office of Scientific Research under award numbers FA9550-17-1-0326, FA9550-21-1-0024, and 12595060.

The context-free version of equality reflection is

$$\textsc{CF-Eq-Reflect}\quad \frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{p} : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}\ \mathsf{by}\ \{\!|\mathsf{p}|\!\}}$$

Note how the assumption set in the conclusion records dependence on p. Instead of recording the entire term p, it suffices to record the assumption set $\{\!|\mathsf{p}|\!\}$, which is obtained by traversing the term p and collecting free variables and metavariables occurring in subterms and in sub-assumption sets of p.

As a consequence of the modification to the structure of equality judgements the conversion rule needs to be adapted.

$$\textsc{CF-Conv-Tm}\quad \frac{\vdash t : A \qquad \vdash A \equiv B\ \mathsf{by}\ \alpha}{\vdash \kappa(t, \alpha \cup \{\!|A|\!\}) : B}$$

The assumption set α used in the derivation of the type equality as well as the assumptions $\{\!|A|\!\}$ used to construct the type A are recorded in the resulting term. In fact, the only places where annotations with assumption sets enter the picture are via conversion terms $\kappa(t, \beta)$ and via proof irrelevant rules such as CF-Eq-Reflect where a premise would go unrecorded in the conclusion without annotation.

Both traditional finitary type theories and context-free type theories enjoy a number of meta-theorems such as admissibility of substitution, presuppositivity, and inversion principles. We prove that strengthening holds for context-free type theories.

To ensure that the formalism of context-free type theories derives the same judgements as traditional finitary type theories up to annotations we provide translations in both directions. Translating from the context-free setting to the traditional is simple enough: move the annotations on metavariables and free variables to metavariable extensions and contexts, elide the conversion terms, and delete the assumption sets. The transformation from traditional theories to context-free theories requires annotation of variables with typing information, insertion of conversions, and reconstruction of assumption sets.

We prove that both of the translations respect derivability.

References

[1] The Andromeda proof assistant. http://www.andromeda-prover.org/.

[2] Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. A general definition of dependent type theories, 2020.

[3] Herman Geuvers, Robbert Krebbers, James McKinna, and Freek Wiedijk. Pure type systems without explicit contexts. *Electronic Proceedings in Theoretical Computer Science*, 34:53–67, 2010.

Clocked Cubical Type Theory

Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi

IT University of Copenhagen, Denmark
(mbkr,mogel,avez@itu.dk)

Guarded recursion [7] is a technique for adding a fix-point combinator to type theory, guarding the recursive call by a modality $\triangleright^\kappa$ to ensure productivity of the unfolding. It has been established that type theories with guarded recursion and multiple clocks allow for encoding coinductive types in type theory [1]. Work in this area has until now focused mainly on basic examples of coinductive types such as streams and the coinductive lifting monad, but much of the interest in coinduction comes from modelling processes, often non-deterministic or probabilistic ones. Representing such coinductive types is difficult because even presenting the functor for which they are final coalgebras involve constructions such as finite powersets that are difficult to represent in type theory. This talk presents a model of a cubical type theory combining multiclocked guarded recursion with higher inductive types (HITs), in which a wide range of coinductive types needed for representing processes can be defined using representations of functors such as the finite powerset functor as HITs. The main technical contribution that makes this encoding work, is the observation that HITs quantified over the object of clocks support an induction principles similar to that original HIT. As a special case we study the type of labelled transition systems and show that path equality for this type coincides with bisimilarity. The work presented in this talk is based on the manuscript [4].

The Type Theory Clocked Cubical Type Theory (CCTT) is an extension of Cubical Type Theory (CTT) [2] with guarded recursion and quantification over clocks. CTT is itself a variant of dependent type theory where the identity type has been replaced by a type of paths, allowing for a computational interpretation of univalence and other extensionality principles.

CCTT also includes guarded recursion using a family of modalities, $\triangleright^\kappa$, indexed by clock labels $\kappa : \mathsf{clock}$. Contexts can be extended with fresh clock labels, and clocks can be abstracted both in types ($\forall\kappa.$) and in terms, in the style of Π types. This allows for a controlled elimination of $\triangleright$ in the form of a type equivalence $\forall\kappa.\, \triangleright^\kappa A \simeq \forall\kappa.\, A$. Using this in combination with guarded recursive types, one can encode coinductive types [1]. For example, to define a type of streams, first define the guarded recursive type of guarded streams satisfying $\mathsf{Str}^\kappa \simeq \mathbb{N} \times \triangleright^\kappa \mathsf{Str}^\kappa$ using the fixed point combinator, then take $\mathsf{Str} \stackrel{\mathsf{def}}{=} \forall\kappa.\mathsf{Str}^\kappa$. Using the fixed point combinator we can show that Str is the final coalgebra of $\mathbb{N} \times -$, and in particular satisfies $\mathsf{Str} \simeq \mathbb{N} \times \mathsf{Str}$.

Combining this with higher inductive types (HITs) we can encode also many of the coinductive types used for modelling non-deterministic processes. For example, recall that the type LTS of finitely branching transition systems with labels in A can be encoded as the final coalgebra of $\mathsf{P_f}(A \times -)$. Here $\mathsf{P_f}$ is the finite powerset functor, which can be defined as a HIT [3]. We can again define LTS by first considering the guarded recursive type LTS^κ satisfying the equation $\mathsf{LTS}^\kappa \simeq \mathsf{P_f}(A \times \triangleright^\kappa \mathsf{LTS}^\kappa)$, and then taking $\mathsf{LTS} \stackrel{\mathsf{def}}{=} \forall\kappa.\mathsf{LTS}^\kappa$. We also need to impose a restriction on the collection labels, namely that the canonical map $A \to \forall\kappa.A$ is an equivalence.

To verify the correctness of these two example encodings, we calculate as follows:

$$\begin{aligned}\mathsf{Str} &\simeq \forall\kappa.(\mathbb{N} \times \triangleright^\kappa \mathsf{Str}^\kappa)\\ &\simeq \forall\kappa.\mathbb{N} \times \forall\kappa.\mathsf{Str}^\kappa\\ &\simeq \mathbb{N} \times \mathsf{Str}\end{aligned} \qquad \begin{aligned}\mathsf{LTS} &\simeq \forall\kappa.\mathsf{P_f}(A \times \triangleright^\kappa \mathsf{LTS}^\kappa)\\ &\simeq \mathsf{P_f}(\forall\kappa.A \times \forall\kappa.\mathsf{LTS}^\kappa)\\ &\simeq \mathsf{P_f}(A \times \mathsf{LTS})\end{aligned}$$

Induction under clock quantification In both of the above cases, the crux of the matter is an interaction of clock quantification and the inductive type formers. To express this in type theory we introduce an induction principle allowing us to consider just elements of the form $\lambda\kappa.\mathsf{con}(\overline{t},\overline{a},\overline{r})$ when constructing a map out of $\forall\kappa.H\,[\kappa]$. For instance, we have the following:

$$\frac{\begin{array}{c}\Gamma, Y : \forall\kappa.\mathsf{P_f}(A\,[\kappa]) \vdash Q(Y)\ \text{type} \qquad \Gamma \vdash X : \forall\kappa.\mathsf{P_f}(A\,[\kappa]) \\ \Gamma, x : \forall\kappa.A\,[\kappa] \vdash u_{\{-\}}(x) : Q(\lambda\kappa.\{x\,[\kappa]\}) \qquad \dots \\ \Gamma, x, x' : \forall\kappa.\mathsf{P_f}(A\,[\kappa]), y : Q(x), y' : Q(x') \vdash u_\cup(x,x',y,y') : Q(\lambda\kappa.x\,[\kappa] \cup y\,[\kappa]) \\ \Gamma, x : \forall\kappa.\mathsf{P_f}(A\,[\kappa]), y : Q(x), i : \mathbb{I} \vdash u_{\mathsf{idem}}(x,y,i) : Q(\lambda\kappa.\mathsf{idem}(x\,[\kappa],i)) \begin{bmatrix}(i=0)\mapsto u_\cup(x,x,y,y)\\(i=1)\mapsto y\end{bmatrix}\end{array}}{\Gamma \vdash \mathsf{ind}^{\mathsf{clock}}_{\mathsf{P_f}}(u_{\{-\}}, u_\cup, u_{\mathsf{idem}}, \dots, X) : Q(X)}$$

Here $\{-\}$ is the singleton set constructor and $\mathsf{idem} : \Pi(x : \mathsf{P_f}(\forall\kappa.\,A\,[\kappa])).\,x \cup x = x$ is the path constructor for idempotency of binary union, and the cases for the remaining constructors are left out. The rule supports the obvious β rules of computing on e.g. input of the form $\lambda\kappa.\{a\,[\kappa]\}$. As a special case, this principle can be used to construct a canonical map $\forall\kappa.\mathsf{P_f}(X) \to \mathsf{P_f}(\forall\kappa.X)$. Similarly, one can construct a map in the opposite direction by $\mathsf{P_f}$-recursion. A second application of these principles then shows that these maps are quasi inverse, meaning in particular that the types are equivalent.

This style of induction is general enough to apply also to truncations and pushouts, proving in a similar fashion that these constructions also commute with quantification over clocks. The case of truncations is particularly important, as it allows us to prove that $\forall\kappa.\exists x : X.\,Q(x) \simeq \exists x : X.\forall\kappa.\,Q(x)$ whenever $X \simeq \forall\kappa.X$. Here $\exists$ is encoded as the composition of Σ and propositional truncation [9]. For example, the notion of bisimilarity for LTS uses the notion of simulation defined as $R(x,y)$ implying

$$\Pi(x' : \mathsf{LTS}, a : A).\,(a,x') \in \mathsf{unfold}(x) \to \exists y' : \mathsf{LTS}.\,((a,y') \in \mathsf{unfold}(y)) \times R(x',y').$$

Bisimilarity can then be defined as a coinductive type via guarded recursion, using the fact that existential quantification over LTS commutes with universal quantification over clocks. We have proved that bisimilarity for LTS coincides with path equality. The proof is based on a similar result for the guarded recursive LTS^κ proved in previous work [6].

Traditionally type theories with guarded recursion implemented as a family of $\triangleright^\kappa$ modalities has included a clock irrelevance axiom. One might wish for such an axiom in our case, since it would free us from the obligation of requiring a clock irrelevant set of labels, and would include the equivalence $\mathbb{N} \simeq \forall\kappa.\mathbb{N}$. Apart from the encoding of coinductive types, the result we have presented can be viewed as a step towards showing that a type theory with this axiom could support many HIT's. This by itself is not sufficient for the encoding theorem however, which mirrors the results in [1].

The model A model of the theory presented here can be constructed in the Orton-Pitts framework [8] for modelling cubical type theory in a topos. Presheaf models of clocked type theory have been constructed previously [5], and such models extend to cubically valued presheaves. To model cubical type theory all type constructors must have associated constructions for composition structures. We have given general conditions that allow for such constructions for operators like $\triangleright^\kappa$. In this model the universal quantification over clocks can be described as a natural number indexed limit, and the structure maps are trivial in the cube component. The fact that HITs are defined in such a way that the kind of constructor cannot be changed by such maps then allows us to validate a general induction principle in the style of the rule above.

References

[1] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. *ACM SIGPLAN Notices*, 48(9):197–208, 2013.

[2] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[3] Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. Finite sets in homotopy type theory. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 201–214. ACM, 2018.

[4] Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. A model of clocked cubical type theory. *arXiv preprint arXiv:2102.01969*, 2021.

[5] Bassel Mannaa, Rasmus Ejlers Møgelberg, and Niccolò Veltri. Ticking clocks as dependent right adjoints: Denotational semantics for clocked type theory. *Logical Methods in Computer Science*, 16, 2020.

[6] Rasmus Ejlers Møgelberg and Niccolò Veltri. Bisimulation as path type for guarded recursive types. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[7] Hiroshi Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science*, pages 255–266. IEEE, 2000.

[8] I. Orton and A.M. Pitts. Axioms for modelling cubical type theory in a topos. *Logical Methods in Computer Science*, Volume 14, Issue 4, Dec 2018.

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

A Formalisation of Approximation Fixpoint Theory

Bart Bogaerts[1*] and Luís Cruz-Filipe[2]

[1] Vrije Universiteit Brussel (VUB), Dept. Computer Science, Brussels, Belgium
bart.bogaerts@vub.be
[2] Univ. Southern Denmark, Dept. Mathematics and Computer Science, Odense, Denmark
lcfilipe@gmail.com

Introduction. *Approximation Fixpoint Theory* (AFT) is an abstract lattice-theoretic framework originally designed to unify semantics of non-monotonic logics [9]. Its first applications were on unifying all major semantics of logic programming [25], autoepistemic logic (AEL) [18], and default logic (DL) [20], thereby resolving a long-standing issue about the relationship between AEL and DL [14, 10, 11]. AFT builds on Tarski's fixpoint theory of monotone operators on a complete lattice [23], starting from the key realisation that, by moving from the original lattice L to the bilattice L^2, Tarski's theory can be generalized into a fixpoint theory for arbitrary (i.e., also non-monotone) operators. Crucially, all that is required to apply AFT to a formalism and obtain several semantics is to define an appropriate approximating operator $L^2 \to L^2$ on this bilattice; the algebraic theory of AFT then directly defines different types of fixpoints that correspond to different types of semantics of the application domain.

In the last decade, AFT has seen several new application domains, including abstract argumentation [22], extensions of logic programming [19, 1, 8, 15], extensions of autoepistemic logic [26], and active integrity constraints [4]. Around the same time, also the theory of AFT has been extended signitificantly with new types of fixpoints [6, 7], and results on *stratification*, [27, 5], *predicate introduction* [28], and *strong equivalence* [24]. All of these results were developed in the highly general setting of lattice theory, making them directly applicable to all application domains, and such ensuring that researchers do not "reinvent the wheel".

Given the success and wide range of applicability of AFT, it sounded natural to formalise this theory in the Coq theorem prover. In this work we report on the first steps of this endeavour.

AFT in a nutshell. AFT studies fixpoints of operators $O\colon L \to L$, where $\langle L, \leq\rangle$ is a lattice, through operators approximating O. These operators work in the *bilattice* $L^2 = \langle L \times L, \leq_p\rangle$, where the *precision order* $\leq_p$ is defined as $(x,y) \leq_p (u,v)$ if $x \leq u$ and $y \geq v$.

Intuitively, a pair $(x,y) \in L^2$ approximates elements in the interval $[x,y] = \{z \in L \mid x \leq z \leq y\}$. We call $(x,y) \in L^2$ *consistent* if $x \leq y$, i.e., if $[x,y]$ is non-empty. The set of consistent elements is denoted by L^c. Pairs (x,x) are called *exact*, since they only approximate x. If (u,v) is consistent and $(x,y) \leq_p (u,v)$, then $[u,v] \subseteq [x,y]$, i.e., (x,y) approximates all elements that (u,v) approximates. We say that (u,v) is *more precise* than (x,y).

An operator $A\colon L^2 \to L^2$ is an *approximator* of O if it is $\leq_p$-monotone and has the property that $A(x,x) = (O(x), O(x))$ for all $x \in L$. As usual in AFT, we often restrict our attention to *symmetric* approximators: approximators A such that, for all x and y, $A(x,y)_1 = A(y,x)_2$. AFT defines the following fixpoints of A in order to study fixpoints of O.

- A *partial supported fixpoint* of A is a fixpoint of A.
- The *Kripke-Kleene fixpoint* of A is the $\leq_p$-least fixpoint of A.
- A *partial stable fixpoint* of A is a pair (x,y) where $x = \mathrm{lfp}(A(\cdot,y)_1)$ and $y = \mathrm{lfp}(A(x,\cdot)_2)$. $A(\cdot,y)_1$ denotes the function $L \to L\colon z \mapsto A(z,y)_1$, and analogously for $A(x,\cdot)_2$.
- The *well-founded fixpoint* of A is the least precise partial stable fixpoint of A.

*This work was partially supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G0B2221N).

Our formalisation. Our aim is to formalise AFT in Coq without using any axioms – in particular, by following a constructive development of AFT. This is a natural choice, since an important motivations for studying fixpoints in computer science is their computability.

Most AFT proofs are by transfinite induction, and we chose to follow the published results closely. We define a type `Ordinal` of (unbounded) ordinals as a record type containing a `Type`, an equivalence relation `eq` (defined equality), a distinguished element `zero`, a successor function `succ` and a strict total order `lt` that is well-founded and compatible with `eq`. We require `succ x` to be the least element strictly greater than `x`, and that we can decide whether an element is of the form `succ x` for some `x`; the elements for which this does not hold are called limits.

Intuitively, the elements of `o:Ordinal` are the ordinals smaller than `o`. For sanity check, we show that the natural numbers form an ordinal, that we can add ω to an ordinal, and that we can build the type of all polynomials in ω (with support `list nat`). If `o:Ordinal`, then we can prove properties of all elements of `o` by transfinite induction. Since we work with defined equality, we need to include a case showing that the property being proved is stable under equality. We do not deal with arithmetic on ordinals, since this is immaterial for our development.

(Complete) lattices are similarly defined as a record type consisting of a carrier type `C` with an equivalence relation, a partial order, and an operator `lub`:(`C` $\rightarrow$ `Prop`) $\rightarrow$ `C` computing least upper bounds. Requiring least upper bounds to be computable restricts the kind of lattices that we can define; still we show that we capture e.g. powerset lattices (which appear in all applications of AFT so far). We also define an operator `BiLattice` : `Lattice` $\rightarrow$ `Lattice`. Given an operator $O : L \rightarrow L$, we inductively define a (O-)chain as a predicate over L that is closed under applications of O and lubs. We prove that, if O is monotonic, then the lub of any chain is an element of the chain, and it is the least fixpoint (lfp) of O (Knaster–Tarski theorem).

AFT provides an alternative characterisation of lfps using so-called O-inductions. Given an operator $O : L \rightarrow L$, $y \in L$ is an O-refinement of x if $x \leq y$ and $y \leq x \vee O(x)$. An O-induction is a transfinite sequence i such that $i_{\eta+1}$ is an O-refinement of i_η and $i_\eta = \text{lub}\{i_{\eta'} \mid \eta' < \eta\}$ for every limit ordinal η. An O-induction is terminal if there is an ordinal η such that the only refinement of i_η is i_η. We prove that every terminal O-induction converges to the least fixpoint of O, and, conversely, that an O-induction that reaches a fixpoint of O is terminal.

Finally, we formalise the notions of approximator and the four types of fixed points defined earlier, and prove their main properties. The whole development can be found at https://doi.org/10.5281/zenodo.4893264.

Discussion. The main challenges encountered so far have to do with our choice to work constructively (without adding any axioms to Coq), which required adapting some proofs in AFT that assume decidability of equality on the lattice.

Several results in AFT require the existence of a "large enough" ordinal for a given lattice. Since we have not been able to construct this ordinal from the lattice, we have defined a notion of "large enough" ordinal, and explicitly add it as a hypothesis when needed. In this way, we choose to accept its existence as a postulate, or prove it using classical logic.

Related work. Transfinite induction has been formalised previously [21, 2, 12, 13, 17], in some cases with a proof of the Knaster–Tarski theorem. Some of these works are based on classical set theory; others formalise ordinals in a way that we found cumbersome to use, which led us to defining our own. The CoLoR library [3, 16] includes a formalisation of the Knaster–Tarski theorem similar to ours. To the best of our knowledge, AFT has not been formalised before.

References

[1] Christian Antic, Thomas Eiter, and Michael Fink. Hex semantics via approximation fixpoint theory. In Pedro Cabalar and Tran Cao Son, editors, *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, volume 8148 of *LNCS*, pages 102–115. Springer, 2013.

[2] Bruno Barras. Sets in Coq, Coq in Sets. *J. Formaliz. Reason.*, 3(1):29–48, 2010.

[3] Frédéric Blanqui and Adam Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.*, 21(4):827–859, 2011.

[4] Bart Bogaerts and Luís Cruz-Filipe. Fixpoint semantics for active integrity constraints. *Artif. Intell.*, 255:43–70, 2018.

[5] Bart Bogaerts and Luís Cruz-Filipe. Stratification in approximation fixpoint theory and its application to active integrity constraints. *ACM Trans. Comput. Log.*, 22(1):6:1–6:19, 2021.

[6] Bart Bogaerts, Joost Vennekens, and Marc Denecker. Grounded fixpoints and their applications in knowledge representation. *Artif. Intell.*, 224:51–71, 2015.

[7] Bart Bogaerts, Joost Vennekens, and Marc Denecker. Safe inductions and their applications in knowledge representation. *Artificial Intelligence*, 259:167 – 185, 2018.

[8] Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs. *CoRR*, abs/1804.08335, 2018. to appear in TPLP.

[9] Marc Denecker, Victor Marek, and Mirosław Truszczyński. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, volume 597 of *The Springer International Series in Engineering and Computer Science*, pages 127–144. Springer US, 2000.

[10] Marc Denecker, Victor Marek, and Mirosław Truszczyński. Uniform semantic treatment of default and autoepistemic logics. *Artif. Intell.*, 143(1):79–122, 2003.

[11] Marc Denecker, Victor Marek, and Mirosław Truszczyński. Reiter's default logic is a logic of autoepistemic reasoning and a good one, too. In Gerd Brewka, Victor Marek, and Mirosław Truszczyński, editors, *Nonmonotonic Reasoning – Essays Celebrating Its 30th Anniversary*, pages 111–144. College Publications, 2011.

[12] Hervé Grall. Proving fixed points. Technical Report hal-00507775, HAL archives ouvertes, 2010.

[13] José Grimm. Implementation of three types of ordinals in Coq. Technical Report RR-8407, INRIA, 2013.

[14] Kurt Konolige. On the relation between default and autoepistemic logic. *Artif. Intell.*, 35(3):343–382, 1988.

[15] Fangfang Liu, Yi Bi, Md. Solimul Chowdhury, Jia-Huai You, and Zhiyong Feng. Flexible approximators for approximating fixpoint theory. In Richard Khoury and Christopher Drummond, editors, *Advances in Artificial Intelligence - 29th Canadian Conference on Artificial Intelligence, Canadian AI 2016, Victoria, BC, Canada, May 31 - June 3, 2016. Proceedings*, volume 9673 of *Lecture Notes in Computer Science*, pages 224–236. Springer, 2016.

[16] Frédéric Blanqui (maintainer). https://github.com/fblanqui/color/blob/master/Util/Relation/Tarski.v. Accessed: 2021-06-02.

[17] Pierre Castéran (maintainer). https://github.com/coq-community/hydra-battles/tree/master/theories/ordinals. Accessed: 2021-06-02.

[18] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[19] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *TPLP*, 7(3):301–353, 2007.

[20] Raymond Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980.

[21] Carlos Simpson. Set-theoretical mathematics in Coq. *CoRR*, abs/math/0402336, 2004.

[22] Hannes Strass. Approximating operators and semantics for abstract dialectical frameworks. *Artif. Intell.*, 205:39–70, 2013.

[23] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 1955.

[24] Mirosław Truszczyński. Strong and uniform equivalence of nonmonotonic theories - an algebraic approach. *Ann. Math. Artif. Intell.*, 48(3-4):245–265, 2006.

[25] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[26] Pieter Van Hertum, Marcos Cramer, Bart Bogaerts, and Marc Denecker. Distributed autoepistemic logic and its application to access control. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 1286–1292. IJCAI/AAAI Press, 2016.

[27] Joost Vennekens, David Gilis, and Marc Denecker. Splitting an operator: Algebraic modularity results for logics with fixpoint semantics. *ACM Trans. Comput. Log.*, 7(4):765–797, 2006.

[28] Joost Vennekens, Maarten Mariën, Johan Wittocx, and Marc Denecker. Predicate introduction for logics with a fixpoint semantics. Parts I and II. *Fundamenta Informaticae*, 79(1-2):187–227, 2007.

Syntax for two-level type theory

Roberta Bonacina[1] and Benedikt Ahrens[2]

[1] Carl Friedrich von Weizsäcker-Zentrum
Universität Tübingen, Tübingen, DE
roberta.bonacina@fsci.uni-tuebingen.de
[2] School of Computer Science
University of Birmingham, Birmingham, UK
b.ahrens@cs.bham.ac.uk

1 Introduction

Homotopy type theory [14] (HoTT) is a vibrant research field in contemporary mathematics. It aims at providing a foundation of mathematics, extending Martin-Löf type theory with the central notions of homotopy level and univalence. These concepts establish a tight connection between types and homotopy spaces. Hence, they allow to formally prove classical results of algebraic topology in a novel, *synthetic* way. Conversely, the connection provides a novel and insightful view of type theory and its models.

The synthetic approach to homotopy theory provided by HoTT is very useful, allowing the results to be invariant under equivalences, but has also drawbacks: properties relying on more than just the related homotopy type cannot be expressed directly in HoTT. An important case is the concept of semisimplicial types, whose definition is so far elusive in HoTT. Voevodsky defined a special Homotopy type system [15] (HTS) as a formal theory in which to discuss constructions that require access to non-homotopy-invariant notions, in particular, the construction of said semisimplicial types. A construction of semisimplicial types in a sort of HTS is described by Herbelin [8].

2 Two-level Type Theory

Two-level type theory [2] (2LTT) is envisioned to be a variant of HTS. 2LTT is composed of two separate levels of types: the outer level is Martin-Löf type theory plus the uniqueness of identity proofs [13] (UIP), which states that all the paths with the same endpoints are equal; the inner level is, in essence, homotopy type theory. These levels are related by a conversion function from the inner to the outer level that preserves context extensions.

The paper "Two-Level Type Theory and Applications" [2] proposes a semantics for 2LTT based on categories with families [7], which justifies reasoning inside the inner system with the full power of homotopy type theory, and reasoning about the inner system within the outer system to circumvent a number of expressive limits of the former. With this approach it is possible to study properties of homotopy type theory syntactically in the two-level system, and, by conservativity [4], to reflect them back in the HoTT world. Among the applications of this approach there are the results on Reedy fibrant diagrams [2], the Univalence Principle [1], and internal ∞-categories with families [9], which have been suggested as a way to overcome known difficulties one encounters when formalising type theory in type theory.

In summary, the motivation and the significance of this approach are that, despite the intrinsic expressive and proving power of homotopy type theory, a wide range of results rely on meta-reasoning and meta-principles, which cannot entirely be formalised within the theory. The two-level approach formalises these meta-principles in a theory which is compatible

both technically and philosophically with homotopy type theory, allowing for their mechanisation. However, the syntax of 2LTT is just sketched in [2] and its proof theory is still largely unexplored.

3 Syntax

In this contribution we propose a system of inference rules for 2LTT with an infinite hierarchy of Tarski-style universes as uniform constructions [11]; the rules allow us to define the syntax in detail, clearly illustrating the behaviour of the two levels, and how they interact. In contrast with [2], we pay particular attention to the definition of Tarski-style universes, following the guidelines of [11]: other than the function El_i, which maps the codes $A : \mathcal{U}_i$ into types $\mathsf{El}_i(A)\,\mathsf{type}$ and is present in [2], we introduce a function lift_i mapping terms of one universe $A : \mathcal{U}_i$ into terms of the next one $\mathsf{lift}_i(A) : \mathcal{U}_{i+1}$. In [2], the lift operation is not present, and the universes are *cumulative*. In our system, those two functions are related by the following rule

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash \mathsf{El}_{i+1}(\mathsf{lift}_i(A)) \equiv \mathsf{El}_i(A)\,\mathsf{type}}\,\mathcal{U}\text{-}\mathsf{lift}$$

stating that El and lift commute. The same happens for inner types; indeed, $A\,\mathsf{type}$ means that A is an outer type, while $A\,\mathsf{type}^o$ means that A is an inner type. This emphasises another difference between our approach and the 2LTT paper: we do not have a *size* for types, whereas in [2] it is specified as $A\,\mathsf{type}_i$ or $A\,\mathsf{type}^o_i$: if $A : \mathcal{U}_i$, then $\mathsf{El}_i(A)\,\mathsf{type}_i$. Moreover, besides the conversion function c from inner to outer types introduced in [2], we define a conversion function c' from inner to outer codes, i.e., terms of the universes: if $A : \mathcal{U}^o_i$, then $c'(A) : \mathcal{U}_i$. It is required that $\mathsf{El}, \mathsf{lift}, c$ and c' commute. We formalise the fact that the conversion function preserves context extension by introducing a notion of equivalence between contexts, which is not present in [2], together with the rule

$$\frac{\Gamma \vdash A\,\mathsf{type}^o}{\Gamma, x : A\,\mathsf{ctx} \equiv \Gamma, y : c(A)\,\mathsf{ctx}}\,\equiv\text{-}\mathsf{ctx}\text{-}\mathsf{EXT}$$

where, if $A\,\mathsf{type}^o$, then $c(A)\,\mathsf{type}$.

Then, we define a generalisation of the notion of category with families which allows us to interpret our formalisation of the two levels and the Tarski-style universes, called *two-level model*, together with a notion of morphism between models. We are aiming to show the compatibility of our system with the (almost) standard semantics for 2LTT by proving an initiality result; this will essentially extend recent work for Martin-Löf type theory by Brunerie, de Boer, Lumsdaine, and Mörtberg [3, 10, 6]. To this end, we define the syntactical two-level model by quotienting the syntax, similar to [12, 5], and aim to prove that it is the initial object in the category of models.

Our long term goal is to develop the basis for a proof assistant that implements 2LTT and allows one to use additional inner and outer axioms, some of which have been already suggested [2], to formalise in parallel the inner and outer levels, and their relations.

References

[1] Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. The Univalence Principle, 2021. https://arxiv.org/abs/2102.06275.

[2] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. 2019. https://arxiv.org/abs/1705.03307v3.

[3] Guillaume Brunerie, Menno de Boer, Peter LeFanu Lumsdaine, and Anders Mörtberg. A formalization of the initiality conjecture in Agda. Talk given by Brunerie at the HoTT 2019 conference, slides available at https://guillaumebrunerie.github.io/pdf/initiality.pdf, 2019.

[4] Paolo Capriotti. *Models of type theory with strict equality*. PhD thesis, University of Nottingham, 2016. http://eprints.nottingham.ac.uk/39382/1/thesis.pdf.

[5] Simon Castellan. Dependent type theory as the initial category with families. http://iso.mor.phis.me/archives/2011-2012/stage-2012-goteburg/report.pdf, 2014.

[6] Menno de Boer. *A Proof and Formalization of the Initiality Conjecture of Dependent Type Theory*. Licentiate dissertation, Department of Mathematics, Stockholm University, 2020.

[7] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 120–134, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[8] Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, 25(5):1116–1131, 2015.

[9] Nicolai Kraus. Internal ∞-Categorical Models of Dependent Type Theory: Towards 2LTT Eating HoTT, 2021. https://arxiv.org/abs/2009.01883.

[10] Peter LeFanu Lumsdaine and Guillaume Brunerie. Initiality for Martin-Löf type theory. Talk at the Homotopy Type Theory Electronic Seminar Talks (HOTTEST), https://www.youtube.com/watch?v=1ogUFFUfU_M, 2020.

[11] Erik Palmgren. On universes in type theory. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998.

[12] Thomas Streicher. *Semantics of Type Theory*. Progress in Theoretical Computer Science. Springer, 1991.

[13] Thomas Streicher. *Investigations into intensional type theory*. Habilitationsschrift, Ludwig-Maximilians-Universität München, 1993.

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

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

On the logical structure of choice and bar induction principles

Nuria Brede and Hugo Herbelin

[1] University of Potsdam, Germany
nuria.brede@uni-potsdam.de
[2] IRIF, CNRS, Université de Paris, Inria, France
hugo.herbelin@inria.fr

We develop an approach to choice principles and to their bar-induction contrapositive principles, as extensionality schemes connecting an "operational" or "intensional" view of respectively ill-foundedness and well-foundedness properties to an "idealistic", "observational" or "extensional" view of these properties. In a first part, we classify and analyse the relations between different intensional definitions of countable ill-foundedness and countable well-foundedness involving Bar Induction, Dependent Choice, Kőnig's Lemma and the Fan Theorem. In a second part, we integrate the Ultrafilter Theorem and the general axiom of choice to the picture and develop, for A a domain, B a codomain and T a predicate "filtering" the finite approximations of functions from A to B, a generic choice axiom GDC_{ABT} and, contrapositively, a generic bar induction principle GBI_{ABT} which uniformly generalise the previously mentioned schemes in the sense that, writing $\mathbb{N}$ and $\mathbb{B}\mathsf{ool}$ for the types of natural numbers and Booleans values respectively, we have:

- $GDC_{ABR_\top}$ intuitionistically captures the strength of the general axiom of choice expressed as $\forall a^A\, \exists b^B\, R(a,b) \to \exists \alpha\, \forall a\, R(a,\alpha(a)))$, where $R_\top$ is an "unconstraining" filter deriving pointwise from the relation R;
- $GDC_{A\mathbb{B}\mathsf{ool}T}$ intuitionistically captures the Boolean Prime Ideal Theorem and Ultrafilter Theorem; contrapositively, $GBI_{A\mathbb{B}\mathsf{ool}T}$ intuitionistically captures Gödel's completeness theorem in the form validity implies provability;
- $GDC_{\mathbb{N}BT}$ intuitionistically captures respectively the axiom of dependent choice; contrapositively, $GBI_{\mathbb{N}BT}$ intuitionistically captures usual bar induction;
- $GDC_{\mathbb{N}\mathbb{B}\mathsf{ool}T}$ captures the choice strength of Weak Kőnig's Lemma, and, up to weak classical reasoning, Weak Kőnig's Lemma itself; contrapositively $GBI_{\mathbb{N}\mathbb{B}\mathsf{ool}T}$ intuitionistically captures the choice strength of the Weak Fan Theorem.

For classifying the countable case, we use the definitions in the next two tables, which all apply to a predicate T over the set A^* of finite sequences of elements of a given domain A. Our focus being purely logical, we do not impose any arithmetical restriction (such as decidability) on the predicate.

We use the letter a to range over elements of A, the letter u to range over the elements of A^*, n to range over the natural numbers $\mathbb{N}$ and α to range over functions from $\mathbb{N}$ to A. The empty sequence is denoted $\langle\rangle$ and sequence extension $u \star a$.

Equivalent concepts on dual predicates	
T is a tree $\forall u\, \forall a\, (u \star a \in T \to u \in T)$	T is monotone $\forall u\, \forall a\, (u \in T \to u \star a \in T)$
T is progressing $\forall u\, (u \in T \to \exists a\, (u \star a \in T))$	T is hereditary $\forall u\, (\forall a\, (u \star a \in T) \to u \in T)$

Dual concepts on dual predicates	
ill-foundedness-style	*well-foundedness-style*
Closure operators	
pruning of T $\nu X.\lambda u.(u \in T \wedge \exists a\,(u \star a \in X))$	hereditary closure of T $\mu X.\lambda u.(u \in T \vee \forall a\,(u \star a \in X))$
Intensional concepts relevant in the general case	
T is a spread $\langle\rangle \in T \wedge T$ progressing	T is barricaded[1] T hereditary $\rightarrow \langle\rangle \in T$
T is productive $\langle\rangle \in$ pruning of T	T is inductively barred $\langle\rangle \in$ hereditary closure of T
Intensional concepts relevant in the case of finite branching	
T has unbounded paths $\forall n\,\exists u\,(\lvert u\rvert = n \wedge \forall v\,(v \leq u \rightarrow v \in T))$	T is uniformly barred $\exists n\,\forall u\,(\lvert u\rvert = n \rightarrow \exists v\,(v \leq u \wedge v \in T))$
T is staged infinite $\forall n\,\exists u\,(\lvert u\rvert = n \wedge u \in T)$	T is staged barred[1] $\exists n\,\forall u\,(\lvert u\rvert = n \rightarrow u \in T)$
Extensional concepts	
T has an infinite branch $\exists \alpha\,\forall u\,(u$ initial segment of $\alpha \rightarrow u \in T)$	T is barred $\forall \alpha\,\exists u\,(u$ initial segment of $\alpha \wedge u \in T)$

[1]not being aware of an established terminology, we use here our own terminology

In their common formulations, choice and bar induction principles (in our case a tree-based form of Dependent Choice, Bar Induction, Kőnig's Lemma and the Fan Theorem) are expressed as connecting one of the intensional concepts (spread, inductively barred, staged infinite and uniformly barred respectively) to the corresponding single extensional concept.

Systematising existing literature, we clarify that the choice and bar induction principles obtained by relying on each of the two intensional concepts relevant to the general case (resp. relevant to the finite case) of a given column are equivalent altogether and that both pairs of pairs (general case and finite case) are equivalent on their common (finite case) intersection.

Then, we take the definitions of inductively barred, productive, having an infinite branch and barred as references and generalise them to the non-necessarily countable case leading to the definitions of GDC_{ABT} and GBI_{ABT} below where $\downarrow T$ and $\uparrow T$ denote respectively the downwards closure by restriction and upwards closure by extension of a predicate T over $(A \times B)^*$ wrt set inclusion, and $\prec$ is about finitely approximating a function.

Dual concepts on dual predicates	
ill-foundedness-style	*well-foundedness-style*
Intensional concepts	
T A-B-approximable from u $\nu X.\lambda u.\,(u \in \downarrow T \wedge\ \forall\, a \notin dom(u)\,\exists b\,(u \star (a,b)) \in X)$	T inductively A-B-barred from u $\mu X.\lambda u.\,(u \in \uparrow T \vee\ \exists\, a \notin dom(u)\,\forall b\,(u \star (a,b)) \in X)$
Extensional concepts	
T has an A-B-choice function $\exists \alpha \in (A \rightarrow B)\,\forall u\,(u \prec \alpha \rightarrow u \in T)$	T is A-B-barred $\forall \alpha \in (A \rightarrow B)\,\exists u\,(u \prec \alpha \wedge u \in T)$

Dual axioms	
Generalised Dependent Choice (GDC_{ABT}) T A-B-approximable from $\langle\rangle$ implies T has an A-B-choice function	Generalized Bar Induction (GBI_{ABT}) T A-B-barred implies T inductively A-B barred from $\langle\rangle$

The generalisations GDC_{ABT} and GBI_{ABT} satisfy the properties given in the introduction but note that having constraints on A, B and T is important. For instance, a diagonalisation argument shows that $GDC_{\mathbb{B}\mathsf{ool}^{\mathbb{N}}\mathbb{N}T}$ and $GBI_{\mathbb{B}\mathsf{ool}^{\mathbb{N}}\mathbb{N}T}$ are inconsistent for a predicate T imposing an injectivity requirement on the choice function.

More details and a bibliography can be found at https://hal.inria.fr/hal-03144849.

Semisimplicial Types in Internal Categories with Families*

Joshua Chen and Nicolai Kraus

University of Nottingham, United Kingdom

Abstract

An open question in homotopy type theory, known as the problem of *constructing semisimplicial types*, is whether one can define a function $\mathsf{SST} : \mathbb{N} \to \mathsf{Type}_1$ such that $\mathsf{SST}(n)$ is the type of all configurations of triangles and tetrahedra of dimension up to n. We show in Agda that semisimplicial types can be constructed in any set-based internal category with families that contains Σ, Π, and a universe. This means that, given a CwF $(\mathsf{Con}, \mathsf{Ty}, \ldots)$ in type theory, we construct a function $\mathsf{SST}_\mathsf{c} : \mathbb{N} \to \mathsf{Con}$.

This project is work in progress, with code available at `github.com/jaycech3n/CwF`.

Semisimplicial Types. *Constructing semisimplicial types* [Uni13, Her15] is an open problem in homotopy type theory and, more generally, in dependent type theory without uniqueness of identity proofs (UIP). It was first discussed between Voevodsky, Lumsdaine and others during the Univalent Foundations special year at the IAS Princeton in 2012–13.

A semisimplicial type of dimension 2 is a tuple (A_0, A_1, A_2), where $A_0 : \mathsf{Type}$ is a type of points, $A_1 : A_0 \to A_0 \to \mathsf{Type}$ is a family of lines (for any two points), and $A_2 : (x\,y\,z : A_0) \to A_1\,x\,y \to A_1\,y\,z \to A_1\,x\,z \to \mathsf{Type}$ is a family of triangle fillers (for any three points and three lines forming a triangle). Similarly, a *semisimplicial type of dimension* n should be a tuple $(A_0, \ldots, A_n)$ which represents families of simplices of dimension at most n. The open problem asks: can one can define a function $\mathsf{SST} : \mathbb{N} \to \mathsf{Type}_1$ such that $\mathsf{SST}(n)$ is equivalent to the (record/Σ-) type of such tuples $(A_0, \ldots, A_n)$?

Construction in Internal CwF's. By an *internal CwF*, we mean a type $\mathsf{Con} : \mathsf{Type}$ together with families $\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Type}$, $\mathsf{Ty} : \mathsf{Con} \to \mathsf{Type}$, $\mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Type}$, and all the components and equalities which are needed to define a category with families [Dyb95]. We say that such a CwF is *set-based* if Con, Ty, Sub, Tm are families of sets in the sense of homotopy type theory, i.e. types satisfying UIP.

The goal of this project is to define, for any set-based internal CwF with Π and Σ-types and a universe U, a function $\mathsf{SST}_\mathsf{c} : \mathbb{N} \to \mathsf{Con}$ in Agda such that $\mathsf{SST}_\mathsf{c}\ n$ is the context $(A_0 : U, A_1 : A_0 \to A_0 \to U, \ldots, A_n : \ldots)$.

Motivation: Connecting Open Problems. A second open problem, originally asked by Shulman [Shu14], is whether homotopy type theory can internalize ("eat" [Cha09]) itself. More concretely, the problem is to formalize the syntax of HoTT inside HoTT as a set-based CwF, and to give interpretation functions which send the syntax to actual types and their elements in the "obvious" way—for example, a context in the CwF should be interpreted as the nested Σ-type of all its components. For a detailed discussion, see the introduction of [Kra21].

Our current project shows how a solution of this question would give rise to a solution to the problem of constructing semisimplicial types, as claimed by Shulman [Shu14]: composing SST_c with the interpretation $\mathsf{Con} \to \mathsf{Type}_1$, we would get the desired function $\mathbb{N} \to \mathsf{Type}_1$.

*This work is supported by the Royal Society, grant reference URF\R1\191055.

Related Work. We are aware of two different scripts which, when given a number n as input, produce valid Agda code for $\mathsf{SST}(n)$—one script using Haskell [Kra14] and one using Python [Bru]. Our function $\mathsf{SST_c}$ can be seen as a dependently typed version of such a script, with Agda replaced by an internal CwF and Haskell/Python replaced by Agda. Since Haskell and Python simply produce strings while $\mathsf{SST_c}$ is required to type-check, the latter is significantly more difficult to define and requires several new ideas.

Our work has analogies with the construction of semisimplicial types in Voevodsky's *homotopy type system* [Voe13] or *2LTT (2-level type theory)* [ACK16, ACKS19]. Here, Agda plays the role of the outer theory[1] and the internal CwF the role of the inner ("fibrant") one. However, our internal CwF is too minimalistic (e.g. does not contain finite types) to mimic the direct construction of [ACK16]. Moreover, 2LTT allows one to first formulate a type in the outer theory and prove its fibrancy afterwards, a strategy which is not possible in our setting.

It is known that semisimplicial *sets* can be constructed in homotopy type theory, i.e. we can define a function $\mathsf{SST}_0 : \mathbb{N} \to \mathsf{Type}_1$ which only considers *sets* of points, *sets* of lines, and so on. Although our CwF's are based on sets, this is unrelated; since our CwF's are not assumed to have an identity type, the notion of truncatedness does not exist for internal types. The fact that Con and $\mathsf{Ty}\,\Gamma$ are sets corresponds to the fact that *judgmental* equality is proof-irrelevant.

Formalization of the Construction. Using the HoTT-Agda library [SH12] we formalize set-based CwF's as records with fields Con, Sub, Ty, Tm, the usual operations thereon, and equations given by their usual presentation as a generalized algebraic theory (see e.g. Fig. 1 of [Kra21]). Since we are motivated by the goal of internalizing constructions in generic homotopy type theory, where many CwF's (such as the formalized syntax proposed by Altenkirch and Kaposi [AK16]) do not satisfy additional definitional equalities, we avoid any use of rewriting pragmas. Our CwF's are further equipped with internal type formers $\hat{\Pi}$ and $\hat{\Sigma}$, as well as a family U of base types (polymorphic over contexts Γ) together with decoding function $\mathsf{el} : \mathsf{Tm}\,\Gamma\,U \to \mathsf{Ty}\,\Gamma$. We then define $\mathsf{SST_c}\,0 \coloneqq U$ and $\mathsf{SST_c}\,(n+1) \coloneqq \big(\mathsf{SST_c}\,n\,,\ (\mathsf{M}\,n \mathrel{\hat{\to}} U)\big)$ by mutual induction with the "matching object" $\mathsf{M} : (n : \mathbb{N}) \to \mathsf{Ty}\,(\mathsf{SST_c}\,n)$, where $\hat{\to}$ is the function type in the internal CwF. The main difficulty lies in defining the type $(\mathsf{M}\,n)$ of $\partial\Delta^{n+1}$-shaped tuples indexed over $\mathsf{SST_c}\,n$.

A high-level description of our approach to this is as follows. We define $\mathsf{M}\,n \coloneqq \mathsf{Sk}\big(n+1, n, \binom{n+2}{n+1}\big)$, where

$$\mathsf{Sk} : (b\,h\,t : \mathbb{N}) \to \mathsf{Ty}\,(\mathsf{SST_c}\,h) \quad \text{for } 0 \le h < b,\ 1 \le t \le \tbinom{b+1}{h+1}$$

encodes, as a nested internal $\hat{\Sigma}$-type $\mathsf{Sk}\,b\,h\,t$, the subfunctor of the representable functor $\Delta_+[b]$[2] which omits all face maps $[i] \to [b]$ for $i > h$, as well as those face maps $[h] \to [b]$ above the t-th (ordered via a bijection $\varphi : \mathsf{Fin}\binom{b+1}{h+1} \cong \Delta_+([h],[b])$). The intuition is that $\mathsf{Sk}\,b\,h\,t$ presents the partial h-dimensional boundary of the b-simplex given by "shape" (b,h,t). The point of this is to allow us to define, by induction on h and t,

$$\mathsf{Sk}\,b\,h\,t \coloneqq \hat{\Sigma}[\sigma : \mathsf{Sk}\,b\,h'\,t']\,\big(A_h(\mathsf{inter}\,\sigma\,\varphi(t))\big),$$

where (h',t') is the lexicographic predecessor of (h,t) and $\mathsf{inter}\,\sigma\,f$ picks out the subtuple of σ corresponding to the face f. This intersection function inter is, again, to be constructed by induction on the indices b, h, t. This is work in progress.

[1] However, we do not assume UIP in Agda, as this would make the connection with *type theory eating itself* impossible. The strictness that seems to be needed to construct semisimplicial types is satisfied in our case because the internal CwF is set-based.

[2] $\Delta_+ \coloneqq \Delta$ without degeneracies, i.e. the category of finite non-empty sets and strictly increasing functions.

References

[ACK16] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending Homotopy Type Theory with Strict Equality. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[ACKS19] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-Level Type Theory and Applications. *ArXiv*, 2019. Available online at https://arxiv.org/abs/1705.03307.

[AK16] Thorsten Altenkirch and Ambrus Kaposi. Type Theory in Type Theory Using Quotient Inductive Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 18–29, New York, NY, USA, 2016. Association for Computing Machinery.

[Bru] Guillaume Brunerie. A Python script generating Agda code for semi-simplicial types, truncated at any given level. Available at https://guillaumebrunerie.github.io/other/semisimplicial.py. Accessed 23 Apr 2021.

[Cha09] James Chapman. Type Theory Should Eat Itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009. Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

[Dyb95] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs (TYPES)*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer-Verlag, 1995.

[Her15] Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, 25(5):1116–1131, 2015.

[Kra14] Nicolai Kraus. A Haskell script to generate the type of n-truncated semi-simplicial types, 2014. Available at https://nicolaikraus.github.io/docs/generateSemiSimp.hs. Accessed 23 Apr 2021.

[Kra21] Nicolai Kraus. Internal ∞-categorical models of dependent type theory: Towards 2LTT eating HoTT, 2021. Available online at https://arxiv.org/abs/2009.01883, to appear in the proceedings of LICS '21.

[SH12] Andrew Swan and the HoTT and UF community. Homotopy type theory, Since 2012. Fork of the original Agda library. Available online at https://github.com/awswan/HoTT-Agda/tree/agda-2.6.1-compatible.

[Shu14] Michael Shulman. Homotopy type theory should eat itself (but so far, it's too big to swallow), 2014. Blog post, https://homotopytypetheory.org/2014/03/03/hott-should-eat-itself.

[Uni13] The Univalent Foundations Program. Semi-simplicial types, 2013. Wiki page of the Univalent Foundations special year at the Institute for Advanced Study, Princeton. Available at https://ncatlab.org/ufias2012/published/Semi-simplicial+types.

[Voe13] Vladimir Voevodsky. A simple type system with two identity types, 2013. Unpublished note available online at https://www.math.ias.edu/vladimir/Lectures.

Intersection Types for a Computational λ-Calculus with Global State

Ugo de'Liguoro[1] and Riccardo Treglia[2]

[1] Università di Torino, C.so Svizzera 185, 10149 Torino, Italy
ugo.deliguoro@unito.it
[2] Università di Torino, C.so Svizzera 185, 10149 Torino, Italy
riccardo.treglia@unito.it

Abstract

We study the semantics of an untyped λ-calculus equipped with operators representing read and write operations from and to a global state. We adopt the monadic approach to model side effects and treat read and write as algebraic operations over a computational monad. We introduce an operational semantics and a type assignment system of intersection types, and prove that types are invariant under reduction and expansion of term and state configurations, and characterize convergent terms via their typings.

Since Strachey and Scott's work in the 60's, λ-calculus and denotational semantics, together with logic and type theory, have been recognized as the mathematical foundations of programming languages. Nonetheless, there are aspects of actual programming languages that have shown to be quite hard to treat, at least with the same elegance as the theory of recursive functions and of algebraic data structures; a prominent case is surely side-effects.

In [Mog91] Moggi proposed a unified framework to reason about λ-calculi embodying various kinds of effects, including side-effects, that has been used by Wadler [Wad92, Wad95] to cleanly implement non-functional aspects into Haskell, a purely functional programming language. Moggi's approach is based on the categorical notion of *computational monad*: instead of adding impure effects to the semantics of a pure functional calculus, effects are subsumed by the abstract concept of "notion of computation" represented by the monad T. To a domain A of values it is associated the domain TA of computations over A, that is an embellished structure in which A can be merged, and such that any morphism f from A to TB extends by a universal construction to a map f^T from TA to TB.

Monadic operations of merging values into computations, i.e. the *unit* of the monad T, and of extension, model how morphisms from values to computations compose, but do not tell anything about how the computational effects are produced. In the theory of *algebraic effects* [PP02, PP03, Pow06], Plotkin and Power have shown under which conditions effect operators live in the category of algebras of a computational monad, which is isomorphic to the category of models of certain equational specifications, namely varieties in the sense of universal algebra [HP07].

In [dT20] we have considered an untyped computational λ-calculus with two sorts of terms: *values* denoting points of some domain D, and *computations* denoting points of TD, where T is some generic monad and $D \cong D \to TD$. The goal was to show how such a calculus can be equipped with an operational semantics and an intersection type system, such that types are invariant under reduction and expansion of computation terms, and convergent computations are characterized by having non-trivial types in the system.

Here, we extend our approach and consider a variant of the *state monad* from [Mog91] and a calculus with two families of operators, indexed over a denumerable set of *locations*: $get_\ell(\lambda x.M)$ reading the value V associated to the location ℓ in the current *state*, and binding x to V in M;

$set_\ell(V, M)$ which modifies the state assigning V to ℓ, and then proceeds as M. This calculus, with minor notational differences, is called *imperative λ-calculus* in [Gav19].

As a first step, we construct a domain D that is isomorphic in a category of domains to $D \to \mathbb{S}\, D$, where $\mathbb{S}$ is the *monad of partiality and state* from [DGL17]. Then, to define the operational semantics, we consider an algebra of states which is parametric in the values, that are denoted by value-terms of the calculus. *State terms* are equated by a theory whose axioms are standard in the literature (see e.g. [Mit96], chap. 6) and are essentially, albeit not literally, the same as those ones for global state in [PP02].

Operational semantics is formalized by the evaluation or *big-step relation* $(M, s) \Downarrow (V, t)$, where M is a (closed) computation, V a value and s, t state-terms. Equivalently, we define in the SOS style a reduction or *small-step relation* $(M, s) \longrightarrow (N, t)$ among pairs of computations and state-terms, which we call *configurations*, such that $(M, s) \Downarrow (V, t)$ if and only if $(M, s) \xrightarrow{*} ([V], t)$, where $[V]$ is the computation trivially returning V.

Types and type assignment system are derived from the domain equation defining D and $\mathbb{S}\, D$, following the method of domain logic in [Abr91]. Type and typing rule definitions are guided along the path of a well understood mathematical method, which indicates both how type syntax and the subtyping relations are constructed and how to shape type assignment rules. The so obtained system is an extension of Curry style intersection type assignment system: see [BDS13] Part III.

The first result we obtain is that in our system types are invariat under reduction and expansion of configurations. This is the key step to the main theorem of this work, that is the characterization of convergence. We say that a program, namely a closed computation term, *converges* if it evaluates to a value and a final state, whatever the initial state is.

In analogy with the lazy lambda-calculus, where a term converges if and only if is typable by $\omega \to \omega$ in a suitable intersection type system, we show that a closed M converges if and only if it is typable by $\omega \to \omega \times \omega$. Consequently, type-checking in our system is undecidable.

The article is available on arXiv: [dT21].

References

[Abr91] S. Abramsky. Domain theory in logical form. *Ann. Pure Appl. Log.*, 51(1-2):1–77, 1991.

[BDS13] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types.* Perspectives in logic. Cambridge University Press, 2013.

[DGL17] U. Dal Lago, F. Gavazzo, and P. B. Levy. Effectful Applicative Bisimilarity: Monads, Relators, and Howe's Method. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017*, pages 1–12. IEEE Computer Society, 2017.

[dT20] U. de'Liguoro and R. Treglia. The untyped computational λ-calculus and its intersection type discipline. *Theor. Comput. Sci.*, 846:141–159, 2020.

[dT21] U. de'Liguoro and R. Treglia. Intersection types for a computational lambda-calculus with global state. https://arxiv.org/abs/2104.01358, 2021.

[Gav19] F. Gavazzo. *Coinductive Equivalences and Metrics for Higher-order Languages with Algebraic Effects.* PhD thesis, University of Bologna, Italy, Aprile 2019.

[HP07] M. Hyland and J. Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electron. Notes Theor. Comput. Sci.*, 172:437–458, 2007.

[Mit96] J.C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1996.

[Mog91] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Pow06] J. Power. Generic models for computational effects. *Theor. Comput. Sci.*, 364(2):254–269, 2006.

[PP02] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FOSSACS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[PP03] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003.

[Wad92] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1992*, pages 1–14. ACM Press, 1992.

[Wad95] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

Abstract of

The Agda Universal Algebra Library and Birkhoff's Theorem in Dependent Type Theory

William DeMeo

Department of Algebra, Charles University, Prague, Czech Republic
williamdemeo@gmail.com

1 Introduction

The Agda Universal Algebra Library (UALib) is a library of types and programs (theorems and proofs) formalizing the foundations of universal algebra in dependent type theory using the Agda programming language and proof assistant. The UALib now includes a substantial collection of definitions, theorems, and proofs from universal algebra and equational logic and as such provides many examples that exhibit the power of inductive and dependent types for representing and reasoning about general algebraic and relational structures.

The first major milestone of the UALib project was the completion of a formal proof of Birkhoff's HSP theorem, which we achieved in January of 2021. To the best of our knowledge, this is the first time Birkhoff's theorem has been formulated and proved in dependent type theory and verified with a proof assistant.

In this presentation of the UALib we discuss some of the more challenging aspects of formalizing universal algebra in type theory and the issues that arise when attempting to constructively prove some of the basic results in that area. In explaining how some of these challenges may be overcome, we hope to demonstrate that dependent type theory and Agda, despite the demands they place on the user, are accessible to working mathematicians who have sufficient patience and a strong desire to formally and constructively codify and verify their work.

After a sobering description of the initial and sometimes painful stage of learning how to do mathematics in type theory and Agda, we hope to make a compelling case for investing in these technologies. Indeed, we are excited to share the gratifying rewards that come with some mastery of type theory and interactive theorem proving.

2 Organization and contributions

In this presentation we limit ourselves to key components of the UALib so that we have space to discuss the more interesting type theoretic and foundational issues that arose when developing these components and completing the first goal of the project (a machine-checked proof of Birkhoff's theorem).[1] We briefly show how *homomorphisms*, *terms*, and *subalgebras* are defined in the UALib and then discuss inductive and dependent types used to represent *free algebras* and *equational classes* of algebras (i.e., *varieties*). Finally, we present the formalization of *Birkhoff's HSP theorem*. The following subsections describe the organization of the presentation in more detail.

[1] A comprehensive overview of the UALib is available in the form of detailed online documentation at https://ualib.org, as well as in the recent series of papers [1, 2, 3].

2.1 The type theory and logical foundations of Agda and UALib

First we recall the logical foundations of Martin-Löf type theory, with special emphasis on the way it is formalized in Agda. After a quick review of Agda's *universe hierarchy*, as well as the classes of *Sigma* and *Pi types*, we discuss a family of concepts that play a vital role in all mechanizations of mathematics using type theory; these are the related concepts of *equality*, *function extensionality*, *proposition extensionality*, *truncation* and *uniqueness of identity proofs*. We then describe our approach to *relation types* and *quotient types*, including our less standard *dependent relation types* which are capable of representing relations of arbitrary arity, defined over arbitrary collections of types.

2.2 Types for algebras, terms, and subalgebras

We describe the basic domain-specific types defined in the UALib. These include types for *algebraic signatures*, *general (universal) algebras*, and *product algebras* (including products over arbitrary classes of algebraic structures), *congruence relations*, and *quotient algebras*. Having thus described the basic objects of study, we focus on relations between them manifested by types for *homomorphisms* and *homomorphic images*. We then define the inductive type of *terms* and mention the formal proof that the term algebra is absolutely free (i.e., it is the initial object in the category of algebras). To conclude this section we define types for *subalgebras*, including an inductive type representing subalgebra generation.

2.3 Types for varieties, free algebras, and Birkhoff's theorem

Here we get to the main technicalities of formalizing nontrivial results in equation logic. We define the "models" relation ($\models$) and the closure operators H, S, and P representing classes of structures that are closed under homomorphic images, subalgebras, and arbitrary products, respectively. We then present the quotient type that represents the (relatively) free algebra.

Finally, we recall the informal statement of Birkhoff's theorem and assemble the components required to formalize that theorem in type theory, and then briefly describe each step in the path to the proof as it is formalized in the UALib.

3 Conclusion

We conclude with the most pressing open questions and considerations for future work on this and related projects. In particular, we discuss the next phase of development in which we will formalize notions of computational complexity theory in the UALib so that we can verify our own results and those of others on the cutting edge of our field of mathematics research.

References

[1] William DeMeo. The Agda Universal Algebra Library, Part 1: Foundation. *CoRR*, abs/2103.05581, 2021. Source code: https://gitlab.com/ualib/ualib.gitlab.io. `arXiv:2103.05581`.

[2] William DeMeo. The Agda Universal Algebra Library, Part 2: Structure. *CoRR*, abs/2103.09092, 2021. Source code: https://gitlab.com/ualib/ualib.gitlab.io. `arXiv:2103.09092`.

[3] William DeMeo. The Agda Universal Algebra Library, Part 3: Identity. *CoRR*, 2021. (to appear) Source code: https://gitlab.com/ualib/ualib.gitlab.io.

A finite-dimensional model for affine, linear quantum lambda calculi with general recursion*†

Alejandro Díaz-Caro[1,3], Malena Ivnisky[2,3],
Hernán Melgratti[2,3], and Benoît Valiron[4]

[1] DCyT, Universidad Nacional de Quilmes. Bernal, Provincia de Buenos Aires, Argentina.
alejandro.diaz-caro@unq.edu.ar
[2] DC, FCEyN, Universidad de Buenos Aires. Buenos Aires, Argentina.
{mivnisky,hmelgra}@dc.uba.ar
[3] CONICET-UBA. Instituto de Ciencias de la Computación (ICC). Buenos Aires, Argentina.
[4] LRI, Centrale Supélec, Université Paris-Saclay. Orsay, France.
benoit.valiron@universite-paris-saclay.fr

Abstract

We introduce a concrete domain model for an extension of the quantum lambda calculus λ_ρ° with a fixpoint operator. A distinctive feature of λ_ρ° is that it relies on density matrices for describing both quantum information and probabilistic distributions over computation states. It has been shown that there is a conservative translation from λ_ρ° to the quantum lambda calculus of Selinger and Valiron. In contrast to existing models for quantum lambda calculi featuring recursion with intuitionistic arrows, our model is finite-dimensional and does not need more than cones of positive matrices and affine arrows.

The design of functional programming languages for quantum computation has roots in the seminal work of Selinger [Sel04], where *quantum flow charts* (QFCs) were introduced. QFCs are (possibly recursive) first-order programs, whose denotational semantics is given in terms of finite-dimensional structures. This approach has been subsequently extended in [SV06] to accommodate higher-order programs. The obtained language, called *quantum lambda calculus*, consists of a typed lambda calculus with quantum data but without recursion. Still, its denotational semantics is given in terms of finite structures. Almost a decade later, a series of works [PSV14, CDVW19, CdV20] have addressed the problem of giving denotational semantics to fixpoint operators for the quantum lambda calculus. Despite their differences, they rely on infinite-dimensional structures to model recursive types and recursion, and functions that can be used repeatedly (i.e., intuitionistic arrows).

In this work we explore a different design choice for adding recursion to quantum lambda calculus that is linear-affine but still has a finite-dimensional model. Concretely, we extend the quantum lambda calculus λ_ρ° [DC17] with a fixpoint operator but without intuitionistic arrows. We remark that λ_ρ° has been shown equivalent [Bor19] to the one proposed by Selinger&Valiron [SV06], but relies on a convenient presentation in terms of density matrices. The denotational semantics of our language follows the approach of Selinger [Sel04]; consequently, we interpret basic types as positive matrices with trace less than or equal to 1. Indeed, a matrix whose trace is 1 represents a terminating program, while a matrix whose trace is smaller than 1 represents a non-terminating program.

Figure 1 summarises the main features of the discussed approaches.

The λ_ρ° calculus has the usual terms of the lambda calculus, i.e. variables, lambda abstractions and application; additionally, it provides constructors for each quantum postulate, namely,

*Partially supported by the French-Argentinian IRP SINFIN and the STIC-AmSud project Qapla'.
†The draft of the full paper can be found at http://mivnisky.github.io/qfixpoint.pdf.

Ref.	Finite	Affine	Duplication	fixpoint	Higher order
[Sel04]	✓			✓	
[SV06]	✓				✓
[PSV14]		✓*	✓	✓	✓
[CDVW19]		✓*	✓	✓	✓
[CdV20]		✓*	✓	✓	✓
Our proposal	✓	✓		✓	✓

* Affinity only for duplicable objects.

Table 1: Summary of related models for the quantum lambda calculus.

(1) ρ^n stands for an n-dimensional quantum state, which is represented by the density matrix ρ; (2) $U^m t$ represents the application of the m-dimensional unitary operator U to the first m qubits of t; (3) $\pi^m t$ corresponds to the measurement of the first m qubits of t; and (4) $t \otimes r$ is the tensor product of t and r. The term $\mathsf{letcase}^\circ\ x = t\ \mathsf{in}\ \{r_0, \ldots, r_{2^n-1}\}$ intuitively stands for a probabilistic distribution over $\{r_0, \ldots, r_{2^n-1}\}$, where the probability associated with the term r_i comes from the measurement denoted by t, i.e., it is assumed that t corresponds to a measurement that may produce 2^n results, each of them with probability p_i. Then, the distribution associated with the $\mathsf{letcase}$ term can be expressed as a generalised density matrix $\sum_i p_i r_i$.

Types for λ°_ρ are given by $A := n \mid (m,n) \mid A \multimap A$, where $n, m \in \mathbb{N}$. The type n stands for the dimension of a density matrix, while (m,n) corresponds to a measurement of m qubits over an n-qubit state (with $m \leq n$), as formally stated by the interpretation function below.

$$[\![n]\!] := \mathcal{D}_n \qquad [\![(m,n)]\!] := \{M \mid M \in \bigoplus_{i=1}^{2^m} \mathcal{D}_n \text{ and } \mathsf{tr}(M) \leq 1\}$$

$$[\![A \multimap B]\!] := \{f \mid f \text{ positive in } ([\![A]\!] \otimes [\![B]\!]) \oplus [\![B]\!]\}$$

We write $\mathcal{D}_n$ for the set of positive matrices of dimension 2^n and trace less than or equal to 1. The operators $\otimes$ and $\oplus$ respectively stand for the tensor product and coproduct of density matrices. The non-standard interpretation of arrows, i.e., $([\![A]\!] \otimes [\![B]\!]) \oplus [\![B]\!]$ instead of $[\![A]\!] \otimes [\![B]\!]$, is motivated by the need of accommodating affine functions, i.e., functions f such that $f(0) \neq 0$. We build upon to the standard Choi representation [Cho75] of a completely positive linear map f as the positive matrix

$$\chi_f := \begin{pmatrix} f(E_{11}) & \ldots & f(E_{1n}) \\ \vdots & & \vdots \\ f(E_{n1}) & \ldots & f(E_{nn}) \end{pmatrix}$$

where $\{E_{ij}\}$ is the canonical basis of $\mathcal{D}_n$, and use the following affine extension

$$\overline{\chi}_f := \begin{pmatrix} f(E_{11}) - f(0_n) & \ldots & f(E_{1n}) - f(0_n) \\ \vdots & & \vdots \\ f(E_{n1}) - f(0_n) & \ldots & f(E_{nn}) - f(0_n) \end{pmatrix} \oplus f(0_n)$$

The left-hand-side expression in the definition of $\overline{\chi}_f$ corresponds to the linear transformation, while the right-hand-side represents the translation.

We show that this finite model is sound and suffices to interpret the fixpoint $\mu x.t$ as the least upper bound of a chain of approximations, i.e., as $\lim_{n\to\infty} \overline{\chi}_f^n(0)$ where $f = \lambda x.t$ and 0 denotes the null matrix.

References

[Bor19] Agustín Borgna. Simulación del lambda cálculo de matrices de densidad en el lambda cálculo cuántico de Selinger y Valiron. Master's thesis, Universidad de Buenos Aires, 2019.

[CdV20] Pierre Clairambault and Marc de Visme. Full abstraction for the quantum lambda-calculus. *Proceedings of the ACM on Programming Languages*, 4 (POPL), 2020.

[CDVW19] Pierre Clairambault, Marc De Visme, and Glynn Winskel. Game semantics for quantum programming. *Proceedings of the ACM on Programming Languages*, 3 (POPL), 2019.

[Cho75] Man-Duen Choi. Completely positive linear maps on complex matrices. *Linear Algebra and its Applications*, 10(3):285–290, 1975.

[DC17] Alejandro Díaz-Caro. A lambda calculus for density matrices with classical and probabilistic controls. In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems (APLAS 2017)*, volume 10695 of *Lecture Notes in Computer Science*, pages 448–467. Springer, Cham, 2017.

[PSV14] Michele Pagani, Peter Selinger, and Benoît Valiron. Applying quantitative semantics to higher-order quantum computing. *SIGPLAN Notices (POPL)*, 49(1):647–658, 2014.

[Sel04] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[SV06] Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

A Formal Theory of Games of Incomplete Information

Hélène Fargier, Érik Martin-Dorel, and Pierre Pomeret-Coquot

IRIT, Université Toulouse III, France
{helene.fargier,erik.martin-dorel,pierre.pomeret}@irit.fr

Abstract

We provide a Coq/SSReflect formalization defining algebraic finite games of incomplete information (generalizing Bayesian games) and proving an extended version of Howson's and Rosenthal's theorem. This is our first step toward a decision theory library in Coq.

1 Introduction

Game theory provides a framework to model and study decision making among several agents. It has been strongly studied in economics, and more recently in artificial intelligence. Formalizing results – that is, writing proofs that can be automatically verified – has been of recent interest in this domain. Existing formalizations in game theory only concern games of complete information (C-Games). We are developing a decision theory library for the Coq proof assistant [7], which may benefit from C-Games-related libraries [12, 2] and/or particular formalized results [13, 15], since a strategic game is a particular form of collective decision. For now, we provide an extensible Coq implementation for finite simultaneous games of incomplete information (I-Games), that is, games in an uncertain environment. We also formalize an extended version of Howson's and Rosenthal's theorem (I-Game to polymatrix C-Game conversion), the proof of which making it possible to validate our definitions.

We adopt an algebraic approach, using abstract utilities, beliefs and expectation operators. Our formalization is able to encompass real-valued (probabilistic) Bayesian games, but also I-Games with partially ordered preferences (e.g. with multicriteria utilities [4, 5]) or based on various uncertainty theories (e.g. possibility theory [9]). We use the Coq feature of dependent typing (e.g. to instantiate agents with different strategy spaces) to stay as general and as close as possible to paper definitions, and work with MathComp/SSReflect library and tactics [14], including its big operators library [3]. Our formalization is freely available on GitHub at `https://ppomco.github.io/coq-incomplete-games-rjcia2021/`.

2 Algebraic Games of Incomplete Information (I-Games)

I-Games model situations where several agents have each to make a choice, without knowing the real state of the world $\omega \in \Omega$. Agents' payoffs depend on their choices, but also on the state of the world. Each agent i has beliefs and preferences, and receives partial information $\theta_i = \tau_i(\omega)$ about the real state of the world. Harsanyi [10] showed one can model those situations depending on agents' possible belief states (usually called "types") without making underlying world states explicit. When knowledge is expressed as a probability and utilities are real-valued, rational agents want to maximize their expected utility – they play a Bayesian game.

There are other ways to express and combine beliefs and preferences. Chu and Halpern [6] algebraically generalize this approach (for a single decision maker), abstracting over domains and expectation operators, which are encapsulated in an evaluation structure $\mathcal{E} = (W, U, V, \oplus, \otimes)$, that we name `eval_struct`. W, U and V are domains for plausibility, utility and "weighted

utility" values, respectively, and are (partially) ordered. $\oplus : V \times V \to V$ and $\otimes : W \times U \to V$ are abstract expectation operators. We require $\oplus$ to be commutative, with a neutral element, to use it as a big operator. We also require V to be decidable (`eqType`). Other conditions on `eval_structs` are only semantic, and are needed neither for definitions, nor for proofs.

Each agent i is modeled by an evaluation structure $\mathcal{E}_i = (W_i, U_i, V_i, \oplus_i, \otimes_i)$, an utility function $u_i : A \times \Theta \to U_i$ ($A = \prod_{i \in N} A_i$ is the space of action profiles, $\Theta = \prod_{i \in N} \Theta_i$ that of belief state profiles), and a plausibility distribution $d_i : \Theta \to W_i$ (providing the confidence on others' beliefs states θ_{-i} given her own belief state θ_i). If $\sigma \in \prod_{i \in N}(\Theta_i \to A_i)$ denotes the action chosen by agents from their belief state, the *Generalized Expected Utility* is defined by:

$$GEU_{(i,\theta_i)}(\sigma) = \bigoplus_{\theta_{-i} \in \Theta_{-i}}{}_i \; d(\theta_{-i} \mid \theta_i) \otimes_i u_i(\sigma^\theta, \theta) \qquad \text{where} \qquad \sigma^\theta(i) = \sigma_i(\theta_i) \text{ for all } i \in N.$$

Finally, an I-Game simply is a dependent structure $G = \big(N, (A_i, \Theta_i, \mathcal{E}_i, d_i, u_i)_{i \in N}\big)$, where N is the finite set of agents (`players`) and, for each agent $i \in N$, $(A_i, \Theta_i, \mathcal{E}_i, d_i, u_i)$ denote her actions, belief states, evaluation structure, plausibility distribution and utility function.

```
Record I_game (player : finType) : Type :=
  { action : player -> finType;
    b_state : player -> finType;
    evalst : player -> eval_struct;
    plausibility : forall i : player, profile b_state -> W (evalst i);
    utility : forall i : player, profile action -> profile b_state -> U (evalst i); }.

Definition GEU : forall (player : finType) (g : I_game player) (i : player),
  b_state g i -> iprofile (b_state g) (action g) -> V (evalst g i).
```

3 Generalizing Howson's and Rosenthal's theorem

Howson's and Rosenthal's theorem [11] provides a polymatrix C-Game with the same Nash equilibria as any 2-players finite Bayesian game. We extend this theorem to any n-players finite I-Games. This constructive proof is quite direct (apart from peculiarities related to the dependently-typed style of the formalization). Our result is more general than Howson's and Rosenthal's one since *a)* it holds for $n \geq 2$ agents, *b)* it deals with any distribution-based decision approach that an `eval_struct` encompasses, and *c)* we state equality of strategy profile's utilities in both games (thus, Nash equilibria correspondence is a corollary – as for any other utility-based concept).

4 Conclusion

We formalized I-Games in the Coq proof assistant, generalizing Bayesian games and enabling to model agents using abstract preferences, beliefs and expectation operators, that is, partially ordered preferences and various uncertainty theories. It enables to instantiate Bayesian games, but also possibilistic games [1] and to devise new incomplete game forms, e.g., multicriteria game. Furthermore, we extended Howson's and Rosenthal's theorem to this algebraic structure and to any number of agents. It gives us confidence in our formalization approach.

As further works, we propose to extend the frame of our model to fuzzy measures (capacities) to encompass belief functions [8, 17] and rank-dependent utility [16] theories. Moreover, we aim to formalize more decision theory results in Coq, in order to build a formal library of algorithmic decision theory, that may benefit from existing works on (complete information) game theory.

References

[1] Nahla Ben Amor, Hélène Fargier, Régis Sabbadin, and Meriem Trabelsi. Possibilistic Games with Incomplete Information. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, pages 1544–1550, 2019.

[2] Alexander Bagnall, Samuel Merten, and Gordon Stewart. A Library for Algorithmic Game Theory in SSReflect/Coq. *Journal of Formalized Reasoning*, 10(1):67–95, 2017.

[3] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical Big Operators. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2008.

[4] David Blackwell. Equivalent Comparisons of Experiments. *The Annals of Mathematical Statistics*, pages 265–272, 1953.

[5] Peter Borm, Freek van Megen, and Stef Tijs. A Perfectness Concept for Multicriteria Games. *Mathematical Methods of Operations Research*, 49(3):401–412, 1999.

[6] Francis C Chu and Joseph Y Halpern. Great Expectations. Part II: Generalized Expected Utility as a Universal Decision Rule. *Artificial Intelligence*, 159(1-2):207–229, 2004.

[7] The Coq Development Team. *The Coq Proof Assistant: Reference Manual: version 8.12*, 2020. URL: https://coq.github.io/doc/V8.12.2/refman/.

[8] Arthur P. Dempster. Upper and Lower Probabilities Induced by a Multivalued Mapping. *The Annals of Mathematical Statistics*, 38:325–339, 1967.

[9] Didier Dubois and Henri Prade. Possibility Theory as a Basis for Qualitative Decision Theory. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95*, pages 1924–1932, 1995.

[10] John C Harsanyi. Games with Incomplete Information Played by "Bayesian" Players, I–III. Part I. The Basic Model. *Management Science*, 14(3):159–182, 1967.

[11] Joseph T Howson Jr and Robert W Rosenthal. Bayesian Equilibria of Finite Two-Person Games with Incomplete Information. *Management Science*, 21(3):313–315, 1974.

[12] Stéphane Le Roux. *Generalisation and Formalisation in Game Theory*. PhD thesis, École normale supérieure de Lyon, 2008.

[13] Stéphane Le Roux, Érik Martin-Dorel, and Jan-Georg Smaus. An Existence Theorem of Nash Equilibrium in Coq and Isabelle. In *Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017*, volume 256 of *Electronic Proceedings in Theoretical Computer Science*, pages 46–60, 2017.

[14] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, January 2021.

[15] Érik Martin-Dorel and Sergei Soloviev. A Formal Study of Boolean Games with Random Formulas as Payoff Functions. In *22nd International Conference on Types for Proofs and Programs, TYPES 2016*, volume 97 of *Leibniz International Proceedings in Informatics*, pages 14:1–14:22, 2016.

[16] John Quiggin. A Theory of Anticipated Utility. *Journal of Economic Behavior & Organization*, 3(4):323–343, 1982.

[17] Glenn Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

Normalization for multimodal type theory

Daniel Gratzer[1]

Aarhus University, Aarhus, Denmark
gratzer@cs.au.dk

To a first approximation, type theory is the study of objects invariant under change of context i.e. fibered connectives. Unfortunately, important features of models of type theory are not fibered and this limits the utility of type theory as an internal language. To manage this contradiction, a number of modal type theories have been proposed which allow for a controlled introduction of non-fibered connectives. Modal type theories are often delicate to construct, which has motivated systems which can be instantiated to different modalities [10, 14].

We focus on one such general modal type theory: MTT, a multimodal dependent type theory [10]. MTT can be instantiated by a strict 2-category specifying a collection of modes, modalities and natural transformations between them—a mode theory—and by altering the input mode theory MTT can be used to reason about e.g., guarded recursion, internalized parametricity, axiomatic cohesion, and the metatheory of type theories [6]. Each instantiation of MTT is known to enjoy certain metatheorems regardless of the mode theory (soundness, canonicity) however, lacking a similarly general normalization result, MTT cannot be implemented in a way which permits easy reuse between different mode theories.

We contribute a normalization algorithm for MTT which applies regardless of its mode theory [9]. As a corollary of this normalization result, we show that type constructors are injective and that conversion in MTT is decidable if and only if equality in the underlying mode theory is decidable. As a further consequence, we show that the typechecking problem for MTT is decidable under these same circumstances. Accordingly, this result provides a theoretical grounding for an implementation which can be instantiated to a variety of modal situations.

Normalization by gluing Our normalization proof follows in the tradition of semantic proofs of normalization and gluing arguments [1–3, 7, 8, 13, 16, 19]. More precisely, we do not proceed by fixing a rewriting system which we prove to be strongly normalizing nor by defining a normalization algorithm on raw terms which is then shown to be sound and complete. Instead, we construct a model in the category given by gluing the syntactic model along a nerve restricting from substitutions to renamings. Normalization follows from this model together with the initiality of syntax. While this approach is standard for normalization-by-gluing proofs, the multimodal apparatus introduces several complications.

MTT Cosmoi In order to discuss the novel features of this proof, we must review the model theory of MTT. For the remainder of this abstract, we consider MTT over a fixed mode theory $\mathcal{M}$. A model of MTT is a strict 2-functor[1] $F : \mathcal{M}^{\mathsf{coop}} \longrightarrow \mathbf{Cat}$ sending $m : \mathcal{M}$ to a category $F(m)$ which supports a model of MLTT. More precisely, in the language of natural models, we require a representable natural transformation $\tau_m : \dot{\mathcal{T}}_m \longrightarrow \mathcal{T}_m$ in $\mathbf{PSh}(F(m))$ closed under the standard connectives of type theory [4]. For each modality $\mu : n \longrightarrow m$, we require a commutative square:

$$\begin{array}{ccc} F(\mu)^*\dot{\mathcal{T}}_n & \longrightarrow & \dot{\mathcal{T}}_m \\ \downarrow & & \downarrow \\ F(\mu)^*\mathcal{T}_n & \longrightarrow & \mathcal{T}_m \end{array}$$

[1]Recall that $\mathcal{M}^{\mathsf{coop}}$ is a 2-category with the same objects as $\mathcal{M}$ but whose 1- and 2-cells are reversed.

The bottom (resp. top) map of this commutative square interprets the formation (resp. introduction) rule of the modal type, see Gratzer et al. [11, Section 5] for further details.

The extra complexity of a 2-functor of models precludes directly constructing the necessary model of MTT. Instead, we begin by generalizing the definition of models following Gratzer and Sterling [12]. We require a pseudofunctor $G : \mathcal{M} \longrightarrow \mathbf{Cat}$ such that $G(m)$ is merely locally Cartesian closed and $G(\mu)$ is a right adjoint, intuitively capturing $\mathbf{PSh}(F(m))$ and $F(m)^*$ respectively. Even in this weaker setting, we can still state all the requirements of a model (a universe τ_m closed under various connectives, etc.) with one exception: we must drop the requirement that each τ_m is fiberwise representable. A pseudofunctor equipped with the remaining applicable structure is an MTT *cosmos*. Note that a model of MTT induces a cosmos, in particular presheaves over contexts induce a cosmos $\mathcal{S}[\![-]\!]$.

We define a category of renamings Ren_m and a functor $\mathbf{i}[m]$ embedding it into the category of contexts Cx_m. Similarly, we define neutral and normal forms together with an embedding into terms such that normal forms correspond to β-normal and η-long terms and show that they form presheaves over Ren_m. Like terms, normal forms contain modalities and 2-cells from $\mathcal{M}$ so, while they are not quotiented by a definitional equality, their equality is decidable if and only if the equality in $\mathcal{M}$ is decidable.

By gluing along $\mathbf{i}[m]^* : \mathbf{PSh}(\mathsf{Cx}_m) \longrightarrow \mathbf{PSh}(\mathsf{Ren}_m)$, we obtain a presheaf topos $\mathcal{G}[\![m]\!]$ and the 2-naturality of $\mathbf{i}[-]$ ensures that $\mathcal{G}[\![-]\!]$ organizes into a 2-functor out of $\mathcal{M}$. The crux of our normalization argument is the construction of a cosmos in $\mathcal{G}[\![-]\!]$ lying over $\mathcal{S}[\![-]\!]$.

The normalization cosmos While $\mathcal{G}[\![m]\!]$ is a presheaf topos, it is cumbersome to manipulate directly. In order to construct the normalization cosmos in $\mathcal{G}[\![-]\!]$, we adapt synthetic Tait computability (STC) [17, 18] to our multimodal setting and work exclusively in the internal language of $\mathcal{G}[\![-]\!]$. Specifically, we show that MTT can be interpreted into $\mathcal{G}[\![-]\!]$ and under this interpretation there is a proposition $\mathbf{syn}_m$ which presents $\mathcal{S}[\![m]\!] = \mathbf{PSh}(\mathsf{Cx}_m)$ (resp. $\mathbf{PSh}(\mathsf{Ren}_m)$) as an open (resp. closed) subtopos of $\mathcal{G}[\![m]\!]$. Having relaxed from models of MTT to cosmoi, the required additional structure can be presented as a sequence of constants to be implemented in the internal language, with the open modality being used to ensure that a connective lies strictly over its counterpart in $\mathcal{S}[\![-]\!]$.

Working internally, we substantiate the constants of an MTT cosmos while ensuring that they lie strictly over their counterparts in $\mathcal{S}[\![m]\!]$. Unlike typical gluing proofs, there is no need to ever exit the internal language and so many subtle constructions are transformed into a sequence of programming exercises in MTT. Following Sterling and Harper [18], we use the internal realignment axiom [5, 15] to ensure that natural constructions of various connectives lie strictly over their counterparts in $\mathcal{S}[\![-]\!]$. We conclude that there is an MTT cosmos in $\mathcal{G}[\![-]\!]$ and a morphism of cosmoi $\pi : \mathcal{G}[\![-]\!] \longrightarrow \mathcal{S}[\![-]\!]$.

The normalization function While syntax $\mathcal{S}[\![-]\!]$ is the initial model of MTT, it is not initial among cosmoi. It still, however, enjoys a privileged position in this category which ensures that each context, term, and type in $\mathcal{S}[\![-]\!]$ is in the essential image of π. From this fact and the definition of π, we conclude the following almost immediately.

Theorem 1. *Each term and type in MTT has a unique normal form.*

As our proof of this fact is constructive, it yields an effective procedure which parallels a familiar normalization-by-evaluation algorithm. Consequently, we obtain the following:

Corollary 2. *The conversion problem in MTT is equivalent to the conversion problem of $\mathcal{M}$.*

Corollary 3. *If modalities and 2-cells enjoy decidable equality, typechecking MTT is decidable.*

References

[1] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. 2001. Normalization by Evaluation for Typed Lambda Calculus with Coproducts. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA) *(LICS '01)*. IEEE Computer Society, 303–. http://dl.acm.org/citation.cfm?id=871816.871869

[2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. [n.d.]. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science* (Berlin, Heidelberg, 1995), David Pitt, David E. Rydeheard, and Peter Johnstone (Eds.). Springer Berlin Heidelberg, 182–199.

[3] Thorsten Altenkirch and Ambrus Kaposi. 2016. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:16. https://doi.org/10.4230/LIPIcs.FSCD.2016.6

[4] Steve Awodey. 2018. Natural models of homotopy type theory. *Mathematical Structures in Computer Science* 28, 2 (2018), 241–286. https://doi.org/10.1017/S0960129516000268 arXiv:1406.3219

[5] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2019. Guarded Cubical Type Theory. *Journal of Automated Reasoning* 63 (2019), 211–253.

[6] Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. 2021. Induction principles for type theories, internally to presheaf categories. arXiv:2102.11649 [cs.LO]

[7] Thierry Coquand. 2019. Canonicity and normalization for dependent type theory. *Theoretical Computer Science* 777 (2019), 184–191. https://doi.org/10.1016/j.tcs.2019.01.015

[8] Marcelo Fiore. 2002. Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Pittsburgh, PA, USA) *(PPDP '02)*. ACM, 26–37. https://doi.org/10.1145/571157.571161

[9] Daniel Gratzer. 2021. Normalization for multimodal type theory. arXiv:2106.01414 [cs.LO]

[10] Daniel Gratzer, G.A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*. ACM. https://doi.org/10.1145/3373718.3394736

[11] Daniel Gratzer, G.A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. Extended version of the LICS paper by the same name.

[12] Daniel Gratzer and Jonathan Sterling. 2020. Syntactic categories for dependent type theory: sketching and adequacy. arXiv:2012.10783 [cs.LO]

[13] Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019. Gluing for type theory. In *Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, Herman Geuvers (Ed.), Vol. 131.

[14] Daniel R. Licata, Michael Shulman, and Mitchell Riley. 2017. A Fibrational Framework for Substructural and Modal Logics. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 25:1–25:22. https://doi.org/10.4230/LIPIcs.FSCD.2017.25

[15] Ian Orton and Andrew M. Pitts. 2018. Axioms for Modelling Cubical Type Theory in a Topos. *Logical Methods in Computer Science* 14, 4 (2018). https://doi.org/10.23638/LMCS-14(4:23)2018 arXiv:1712.04864

[16] Michael Shulman. 2015. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science* 25, 5 (2015), 1203–1277. https://doi.org/10.1017/S0960129514000565 arXiv:1203.3253

[17] Jonathan Sterling. 2021. First Steps in Synthetic Tait Computability. (2021). Thesis draft.

[18] Jonathan Sterling and Robert Harper. 2020. Logical Relations As Types: Proof-Relevant Parametricity for Program Modules. (2020). arXiv:2010.08599 https://arxiv.org/abs/2010.08599 Draft paper.

[19] Thomas Streicher. 1998. Categorical intuitions underlying semantic normalisation proofs. In *Preliminary Proceedings of the APPSEM Workshop on Normalisation by Evaluation*, O. Danvy and P. Dybjer (Eds.). Department of Computer Science, Aarhus University.

Modalities and Parametric Adjoints

Daniel Gratzer,[1] Evan Cavallo,[2] G.A. Kavvos,[3] Adrien Guatto,[4] Lars Birkedal[1]

[1] Aarhus University
gratzer@cs.au.dk, birkedal@cs.au.dk
[2] Stockholm University
evan.cavallo@math.su.se
[3] University of Bristol
alex.kavvos@bristol.ac.uk
[4] Université de Paris, CNRS, IRIF
guatto@irif.fr

Recently, a line of modal type theories centering on 'Fitch style' modalities has been proposed [1, 3, 6, 10]. These type theories incorporate a non-fibered modality which behaves like a right adjoint. Specifically, Fitch style type theories pair a modality $\Box$ with a functor on contexts $🔒$ to form a *dependent adjunction*, whose transpositions constitute the introduction and elimination rules:

$$\frac{\Gamma.🔒 \vdash M : A}{\Gamma \vdash \mathsf{mod}(M) : \Box A} \qquad \frac{\Gamma \vdash M : \Box A}{\Gamma.🔒 \vdash \mathsf{unmod}(M) : A}$$

Requiring that these operations form a bijection provides β and η rules for $\Box$. Moreover, the introduction rule is evidently stable under substitution; categorically, this is the the naturality of the bijection in Γ. Unfortunately, the same cannot be said for the elimination principle $\mathsf{unmod}(-)$. A type theorist will immediately identify the "non-general" context in the conclusion and worry that it will prove impossible to commute an arbitrary substitution past $\mathsf{unmod}(-)$. To address this, prior Fitch style type theories have adopted slight variations on the rule, each baking in the bare minimum to ensure the admissibility of substitution.

While it provides a convenient syntax, this approach is brittle, with each modification to the modal apparatus requiring a full redesign. Even restricting attention to a single modal type theory, the resultant syntax cannot be used effectively as an internal language: the proof of admissibility of substitution requires induction not just on terms, but on the definable substitutions. When we use the calculus as an internal language, we add in additional substitutions from the model to more effectively capture the particulars of this situation. In so doing, however, we disrupt the substitution property of our type theory: a lemma proved in one context can no longer be freely applied in a different context, resulting in a type theory that is much less useful. While other solutions to this problem have been proposed, most notably a weakening of the elimination rule [9], it has remained unknown how to combine even two common Fitch style modalities such as $\Box$ and $\triangleright$ [7] in one dependent type theory.

We address this state of affairs by assuming additional structure, that of a parametric adjunction, which reconciles the strong Fitch style elimination rule with substitution. We thereby contribute FitchTT, a modal type theory which can support an arbitrary collection of Fitch style modalities and natural transformations between them [8]. It is a small step from one parametric adjoint modality to full FitchTT, but this is testament to the utility of parametric adjoints in structuring the theory. Indeed, FitchTT is capable of containing multiple interacting modalities such as the aforementioned $\Box$ and $\triangleright$ without the difficulties of prior approaches.

More than this, the extra structure of parametric adjoints is latent in all prior Fitch calculi, and their presence in the initial models of these type theories accounts for the admissibility of substitution. As a result, FitchTT conservatively extends DRA [3] and embeds in $\mathsf{MLTT}_{🔒}$ [10].

Furthermore, this extra structure allows us to systematically rederive the syntax of a single-clock variant of Clocked Type Theory [1] and parametric type theory [4] in a uniform setting.

The special case of functions To motivate the role of parametric adjoints in Fitch style modalities, we focus on a concrete modality: exponentiation by a closed type $\mathfrak{C}$. Specializing the above rules with $\Box A = \mathfrak{C} \to A$ and $\Gamma.🔒 = \Gamma.\mathfrak{C}$, we see that the introduction rule is the familiar introduction rule of dependent products, but the elimination rule is more surprising:

$$\frac{\Gamma \vdash M : \mathfrak{C} \to A}{\Gamma.\mathfrak{C} \vdash \mathsf{unmod}(M) : A}$$

This rule is equivalent to the application rule because $[\Delta, \Gamma.\mathfrak{C}] \cong [\Delta, \Gamma] \times [\Delta, \mathbf{1}.\mathfrak{C}]$. We first bundle a substitution $r \colon \Delta \longrightarrow \mathbf{1}.\mathfrak{C}$ with Δ and view the pairing as an object in the slice category $\mathbf{Cx}/\mathfrak{C}$. By taking $\Gamma.\mathfrak{C}$ as another object over $\mathbf{1}.\mathfrak{C}$ by projection, we can rewrite this isomorphism in a more compact form:

$$[\Delta, \Gamma]_{\mathbf{Cx}} \cong [(\Delta, r), (\Gamma.\mathfrak{C}, \mathbf{v}_k)]_{\mathbf{Cx}/\mathfrak{C}}$$

Written this way, we see that $-.\mathfrak{C}$ is a right adjoint, not as a functor $\mathbf{Cx} \longrightarrow \mathbf{Cx}$ but as a functor $\mathbf{Cx} \longrightarrow \mathbf{Cx}/\mathfrak{C}$. More concisely, $-.\mathfrak{C}$ is a parametric right adjoint (PRA):

Definition 1. $F \colon \mathcal{C} \longrightarrow \mathcal{D}$ is a parametric right adjoint if $F/\mathbf{1} : \mathcal{C} \longrightarrow \mathcal{D}/F(\mathbf{1})$ is a right adjoint.

In the case of $-.\mathfrak{C}$, the left adjoint U is the forgetful functor $\mathbf{Cx}/\mathfrak{C} \longrightarrow \mathbf{Cx}$ which sends (Γ, r) to Γ. We now restate the traditional application rule purely in terms of this parametric adjunction:

$$\frac{r \colon \Gamma \longrightarrow \mathbf{1}.\mathfrak{C} \qquad U(\Gamma, r) \vdash M : \mathfrak{C} \to A}{\Gamma \vdash M\langle r \rangle : A[\eta[r]]}$$

Unlike the rule for $\mathsf{unmod}(-)$ specialized to $\mathfrak{C} \to -$, this rule is stable under substitution. Recalling that $U(\Gamma, r) = \Gamma$, this rule becomes precisely the familiar application rule.

Generalizing with PRAs Taking our cue from this special example, we consider a general Fitch style modality $\langle \mu \,|\, - \rangle$ whose left adjoint on contexts $-.\{\mu\}$ is a parametric right adjoint. We adopt the notation $\Gamma/(r : \mu) = U(\Gamma, r)$ for the parametric left adjoint to $-.\{\mu\}$ by analogy with the construct used in nominal and parametric type theories [2, 4, 5].

The introduction rule for $\langle \mu \,|\, - \rangle$ remains unchanged, but we now take the modified variant of the application rule for our elimination rule:

$$\frac{r \colon \Gamma \longrightarrow \mathbf{1}.\{\mu\} \qquad \Gamma/(r : \mu) \vdash M : \langle \mu \,|\, A \rangle}{\Gamma \vdash M \mathbin{@} r : A[\eta[r]]}$$

We may equip this rule with β and η rules which closely mirror those of dependent products. We further observe that $M \mathbin{@} r$ is interderivable with $\mathsf{unmod}(M)$, but stable under substitution.

Unlike the ad hoc variants of $\mathsf{unmod}(-)$ used in prior Fitch style type theories, this rule scales to multiple modalities. In fact, no issues arise if we allow any strict 2-category of modes, modalities, and natural transformations [11] between them, provided that we require that each modality is equipped with a left adjoint on contexts which is itself a PRA.

References

[1] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The clocks are ticking: No more delays!. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE. https://doi.org/10.1109/LICS.2017.8005097

[2] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. 2015. A Presheaf Model of Parametric Type Theory. *Electronic Notes in Theoretical Computer Science* 319 (2015), 67–82. https://doi.org/10.1016/j.entcs.2015.12.006

[3] Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2020. Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science* 30, 2 (2020), 118–138. https://doi.org/10.1017/S0960129519000197 arXiv:1804.05236

[4] Evan Cavallo and Robert Harper. 2020. Internal Parametricity for Cubical Type Theory. In *28th EACSL Annual Conference on Computer Science Logic (CSL 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 152)*, Maribel Fernández and Anca Muscholl (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:17. https://doi.org/10.4230/LIPIcs.CSL.2020.13

[5] James Cheney. 2012. A dependent nominal type theory. *Log. Methods Comput. Sci.* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:8)2012

[6] Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, 258–275.

[7] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2015. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *Foundations of Software Science and Computation Structures* (Berlin, Heidelberg), Andrew Pitts (Ed.). Springer Berlin Heidelberg, 407–421.

[8] Daniel Gratzer, Evan Cavallo, G.A. Kavvos, Adrien Guatto, and Lars Birkedal. 2021. Modalities and Parametric Adjoints. https://jozefg.github.io/papers/modalities-and-parametric-adjoints.pdf Under review.

[9] Daniel Gratzer, G.A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*. ACM. https://doi.org/10.1145/3373718.3394736

[10] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a Modal Dependent Type Theory. *Proc. ACM Program. Lang.* 3 (2019). Issue ICFP. https://doi.org/10.1145/3341711

[11] Daniel R. Licata and Michael Shulman. 2016. Adjoint Logic with a 2-Category of Modes. In *Logical Foundations of Computer Science*, Sergei Artemov and Anil Nerode (Eds.). Springer International Publishing, 219–235. https://doi.org/10.1007/978-3-319-27683-0_16

Proof terms for generalized classical natural deduction

Herman Geuvers[1] and Tonny Hurkens

[1] Radboud University Nijmegen & Technical University Eindhoven (NL)
herman@cs.ru.nl
[2] hurkens@science.ru.nl

We build on the method of deriving natural deduction rules for a connective c from the truth table t_c of c, as it has been introduced in [2, 3]. In [2] we defined the method for both constructive logic and classical logic, and the constructive case has been studied in more detail, using proof terms for deductions, in [3, 4]. Here we focus on the classical case: we introduce proof terms for the classical natural deduction rules that we extract from a truth table and we use them to study *normal deductions*. These normal deductions, or *deductions in normal form*, should satisfy the *sub-formula property*: every formula that occurs in a normal deduction is a sub-formula of the conclusion or one of the assumptions. We prove this property by giving a *normalization procedure*, that transform a deduction into one in normal form.

A special advantage of our general method of extracting deduction rules from the truth table is that our deduction rules have a specific format. (The format is somewhat related to the generalized elimination/introduction rules defined in [5]. In [3] we discuss the connections in more detail.) This also allows us to give a generic format for the proof terms and to study them generally for an arbitrary set of connectives. Our method also has the advantage (which has already been discussed in [2]), that we can study connectives "in isolation": e.g. there are classical rules for implication, which use only implication (and no negation). In case a connective c is *monotonic*, the constructive and the classical deduction rules are equivalent, but in case c is *non-monotonic* they are not. (Connective c of arity n is monotonic iff its truth table function $t_c : \{0,1\}^n \to \{0,1\}$ is monotonic with respect to the ordering induced by $0 \leq 1$.) We will also show that, for a logic with connectice set $\mathcal{C}$, if one non-monotonic connective $c \in \mathcal{C}$ has classical rules, then all connectives in $\mathcal{C}$ "become" classical. So, e.g. if we have $\{\to, \neg\}$ as connectives with classical rules for $\to$ and constructive rules for $\neg$, we can derive the classical rules for $\neg$.

To be more precise: the elimination rules are as follows. If t_c is the truth table for the n-ary connective c, and $t_c(a_1, \ldots, a_n) = 0$ we have the following rule, where $\Phi = c(A_1, \ldots, A_n)$ is the formula we eliminate. Here, the A_i correspond to the 1-entries and the A_j correspond to the 0-entries of the row $(a_1, \ldots, a_n)$ in t_c that we consider. The formula D is arbitrary.

$$\frac{\vdash \Phi \quad \vdash A_{i_1} \quad \ldots \quad \vdash A_{i_k} \qquad A_{j_1} \vdash D \quad \ldots \quad A_{j_\ell} \vdash D}{\vdash D}\ \mathrm{el}$$

So, $A_{i_1}, \ldots, A_{i_k}, A_{j_1}, \ldots, A_{j_\ell}$ are the direct sub-formulas of $\Phi = c(A_1, \ldots, A_n)$ and we refer to the A_i as *lemma* and the A_j as *casus* in the derivation rule. The classical introduction rules are derived from rows in t_c with $t_c(a_1, \ldots, a_n) = 1$. They are as follows, where again the A_i correspond to the 1-entries and the A_j correspond to the 0-entries of the row $(a_1, \ldots, a_n)$.

$$\frac{\Phi \vdash D \quad \vdash A_{i_1} \quad \ldots \quad \vdash A_{i_k} \qquad A_{j_1} \vdash D \quad \ldots \quad A_{j_\ell} \vdash D}{\vdash D}\ \mathrm{in}$$

So, for a connective c, every line in the truth table t_c gives a deduction rule: an elimination rule if $t_c(a_1, \ldots, a_n) = 0$ and an introduction rule if $t_c(a_1, \ldots, a_n) = 1$, which yields 2^n rules, which can be optimized. For further details, see [2, 3].

Given a logic with classical derivation rules as derived from truth tables for a set of connectives $\mathcal{C}$, we can define the typed λ-calculus $\lambda^{\mathcal{C}}$, which has judgments $\Gamma \vdash t : A$, where A is a formula, Γ is a set of declarations $\{x_1 : A_1, \ldots, x_m : A_m\}$, where the A_i are formulas and the x_i are term-variables such that every x_i occurs at most once in Γ, and t is a *proof-term*. The abstract syntax for proof-terms, Term, is as follows, where x ranges over variables.

$$t ::= x \mid (\lambda y : A.t) \star \{\overline{t} \,;\, \overline{\lambda x : A.t}\} \mid t \cdot [\overline{t} \,;\, \overline{\lambda x : A.t}]$$

The terms are *typed* using the following derivation rules, where the first rule is the *axiom* rule basically stating that $\Gamma \vdash A$ if $A \in \Gamma$.

$$\frac{}{\Gamma \vdash x_i : A_i} \text{ if } x_i : A_i \in \Gamma$$

$$\frac{\Gamma \vdash t : \Phi \quad \ldots \quad \Gamma \vdash p_k : A_k \quad \ldots \quad \Gamma, y_\ell : A_\ell \vdash q_\ell : D \quad \ldots}{\Gamma \vdash t \cdot [\overline{p} \,;\, \overline{\lambda y : A.q}] : D} \text{ el}$$

$$\frac{\Gamma, z : \Phi \vdash t : D \quad \ldots \quad \Gamma \vdash p_i : A_i \quad \ldots \quad \Gamma, y_j : A_j \vdash q_j : D \quad \ldots}{\Gamma \vdash (\lambda z : \Phi.t) \star \{\overline{p} \,;\, \overline{\lambda y : A.q}\} : D} \text{ in}$$

Here, $\overline{p}$ is the sequence of terms $p_1, \ldots, p_{m'}$ for all the 1-entries in the truth table, and $\overline{\lambda y : A.q}$ is the sequence of terms $\lambda y_1 : A_1.q_1, \ldots, \lambda y_m : A_m.q_m$ for all the 0-entries in the truth table.

To reduce the proof terms (and thereby the deductions) to normal form, we first perform *permutation reductions* and then we eliminate *detours*. This is similar to the constructive case, except for now

- a term is in *permutation normal form* if all lemmas are variables,
- a *detour* is an elimination of Φ followed by an introduction of Φ.

Note the difference with constructive logic, where a detour is an introduction directly followed by an elimination. Here it is the other way around, and the introduction need not follow the elimination directly.

We can be more precise by giving the following abstract syntax N for *permutation normal forms*:

$$N ::= x \mid (\lambda y : A.N) \star \{\overline{z} \,;\, \overline{\lambda x : A.N}\} \mid y \cdot [\overline{z} \,;\, \overline{\lambda x : A.N}],$$

where x, y, z range over variables. We can obtain a deduction in permutation normal form by moving applications of an elimination or introduction rule that have a non-trivial lemma upwards, until all lemmas become trivial: the proof-terms are variables. (This only works for the classical case!) Now, a *detour* is a pattern of the following shape

$$(\lambda x : \Phi. \ldots (x \cdot [\overline{v} \,;\, \overline{\lambda w : B.s}]) \ldots) \star \{\overline{z} \,;\, \overline{\lambda y : A.q}\}$$

that is, an elimination of Φ followed by an introduction of Φ, with an arbitrary number of steps in between. For terms in permutation normal form, we show how detours can be eliminated, obtaining a term in normal form which satisfies the sub-formula property. It should be noted that in the above case, this need not be the only occurrence of x (one $\lambda x : \Phi$ may give rise to several detours), so this elimination is not straightforward. Another situation is that x may not occur at all; that is the simplest situation and the sub-term $(\lambda x : \Phi.t) \star \{\overline{z} \,;\, \overline{\lambda y : A.q}\}$ can simply be replaced by t.

We also study how our reduction of classical derivations relates to well-known reductions of classical natural deduction, like as in [1], and how we can interpret control operators.

References

[1] Z. Ariola and H. Herbelin. Minimal classical logic and control operators. In *ICALP*, volume 2719 of *LNCS*, pages 871–885. Springer, 2003.

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

[3] H. Geuvers and T. Hurkens. Proof Terms for Generalized Natural Deduction. In A. Abel, F. Nordvall Forsberg, and A. Kaposi, editors, *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*, volume 104 of *LIPIcs*, pages 3:1–3:39, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] H. Geuvers, I. van der Giessen, and T. Hurkens. Strong normalization for truth table natural deduction. *Fundam. Inform.*, 170(1-3):139–176, 2019.

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

On the Use of Isabelle/HOL for Formalizing New Concise Axiomatic Systems for Classical Propositional Logic

Asta Halkjær From and Jørgen Villadsen

Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

We describe novel axiomatic systems for classical propositional logic: one based on the K and S combinators and elimination rules and one on transitivity of implication, explosion and rules for disjunction. We show how Isabelle/HOL helps investigate such systems.

Introduction

We have recently [7] formalized axiomatic systems for (functionally complete) fragments of classical propositional logic in Isabelle/HOL [10] and build on this work here. The first fragment ("System W", 667 lines) is based on falsity and implication, and we gave a traditional presentation of three different historical axiom systems [2]. The second ("System R", 629 lines, notable overlap with System W) is based on negation and disjunction. We made similar investigations for the two systems, in particular into soundness and completeness and the independence of certain axioms. Here, we consider new concise axioms ("TYPES1" and "TYPES2", each about 30 lines) for the first fragment and use Isabelle/HOL to relate them to the existing work. From a teaching perspective, the proof assistant provides an environment for students where they can confidently explore variations of the systems. The formalizations are available online:

https://people.compute.dtu.dk/ahfrom/types2021/

In a way, our explorations follow in historical footsteps:

> *Undoubtedly there was a competitive element to the search for ever better axiom systems, in particular in the attempt to find single axioms for various systems, and the exercise has been smiled upon or even belittled as a mere "sport"...*
>
> *Tarski showed in 1925 that the pure implicational calculus could be based on a single axiom, but a series of improvements by Wajsberg and Łukasiewicz led to the latter's discovering in 1936 that the formula CCCpqrCCrpCsp could serve as single axiom and that no shorter axiom would suffice, though the publication of this result had to wait until 1948.*
>
> Source: https://plato.stanford.edu/entries/lukasiewicz/

Our Approach and Related Work

We deeply embed the logic by declaring the syntax as a datatype, which the semantics interprets into the higher-order logic. We formalize our proof systems as inductive predicates based on modus ponens and various axioms. Soundness, while trivial, is important and the proof obligations can be discharged automatically. Completeness follows the Henkin style of building models from maximal consistent sets of formulas (described elsewhere [3]). Both properties are shown for other systems by translating derivations (the "sledgehammer" tool can help).

In contrast to Michaelis and Nipkow [8,9] we establish the soundness and completeness of a number of different axiomatic systems and the two presented here are novel as far as we know.

The TYPES1 Theory

The first two axioms correspond to the combinators K and S (involving only implication), and the second two axioms eliminate the negation and disjunction operators, respectively.

```
inductive H :: ⟨form ⇒ bool⟩ (⟨⊢ -⟩ [50] 50) where
  ⟨⊢ p ⟹ ⊢ (p ⟶ q) ⟹ ⊢ q⟩ |
  ⟨⊢ (p ⟶ q ⟶ p)⟩ |
  ⟨⊢ ((p ⟶ q ⟶ r) ⟶ (p ⟶ q) ⟶ p ⟶ r)⟩ |
  ⟨⊢ (p ⟶ ¬ p ⟶ q)⟩ |
  ⟨⊢ ((p ⟶ r) ⟶ (q ⟶ r) ⟶ p ∨ q ⟶ r)⟩
```

The last axiom is classical since disjunction is an abbreviation: $p \vee q \equiv \neg\ p \longrightarrow q$. With Isabelle/HOL we quickly prove other formulas derivable (also using Sledgehammer):

```
proposition ∗: ⟨⊢ (¬ ¬ p ⟶ p)⟩
  by (metis H.intros)
```

This formula is the key lemma in the completeness proof for the axiomatic system TYPES1.

The TYPES2 Theory

The second axiom system is based on transitivity of implication, the principle of explosion, idempotence of disjunction as well weakening on the left-hand side.

```
inductive H :: ⟨form ⇒ bool⟩ (⟨⊢ -⟩ [50] 50) where
  ⟨⊢ p ⟹ ⊢ (p ⟶ q) ⟹ ⊢ q⟩ |
  ⟨⊢ ((p ⟶ q) ⟶ (q ⟶ r) ⟶ p ⟶ r)⟩ |
  ⟨⊢ (p ∧ ¬ p ⟶ q)⟩ |
  ⟨⊢ (p ∨ p ⟶ p)⟩ |
  ⟨⊢ (p ⟶ q ∨ p)⟩
```

Unabbreviated, idempotence of disjunction is the classical principle $(\neg\ p \longrightarrow p) \longrightarrow p$. We automatically show derivability of Peirce's law, the K combinator and a formulation of the principle of explosion based on the primitive falsity (all proofs by *metis H.intros*):

```
lemma ∗: ⟨⊢ (((p ⟶ q) ⟶ p) ⟶ p)⟩
lemma ∗∗: ⟨⊢ (p ⟶ q ⟶ p)⟩
lemma ∗∗∗: ⟨⊢ (⊥ ⟶ p)⟩
```

Like for the axiomatic system TYPES1 we prove completeness for TYPES2 by identifying the key lemmas required — so far we have more or less used a trial and error approach.

Conclusions and Future Work

We have demonstrated how Isabelle/HOL can be used to formalize new concise axiomatic systems for classical propositional logic. The automation, including Sledgehammer, helps relate the systems to existing work by proving the derivability of key formulas. The formalization includes verification that Łukasiewicz's formula *CCCpqrCCrpCsp* can indeed serve as single axiom [7], a task that Pfenning failed to complete [11, 13] and Wos and Fitelson did in OTTER [12].

Our work is part of the IsaFoL project, Isabelle Formalization of Logic [1], embracing first- and higher-order logic too, which aims "to develop libraries and a methodology to support modern research in automated reasoning" [1]. In addition to new axiomatic systems for (classical) propositional logic we see extensions to modal logics [4–6] as a promising line of future work.

References

[1] Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 1–13. ACM, 2019.

[2] Alonzo Church. *Introduction to Mathematical Logic*. Princeton: Princeton University Press, 1956.

[3] Asta Halkjær From. Formalizing Henkin-style completeness of an axiomatic system for propositional logic. In *WeSSLLII + ESSLLI Virtual Student Session*, 2020. https://www.brandeis.edu/nasslli2020/pdfs/student-session-proceedings-compressed.pdf#page=8, preliminary paper, accepted for post-proceedings.

[4] Asta Halkjær From, Patrick Blackburn, and Jørgen Villadsen. Formalizing a Seligman-style tableau system for hybrid logic - (short paper). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 474–481. Springer, 2020.

[5] Asta Halkjær From. Epistemic logic. *Archive of Formal Proofs*, October 2018. https://isa-afp.org/entries/Epistemic_Logic.html, Formal proof development.

[6] Asta Halkjær From. Formalizing a Seligman-style tableau system for hybrid logic. *Archive of Formal Proofs*, December 2019. https://isa-afp.org/entries/Hybrid_Logic.html, Formal proof development.

[7] Asta Halkjær From, Agnes Moesgård Eschen, and Jørgen Villadsen. Formalizing axiomatic systems for propositional logic in Isabelle/HOL. To appear. http://people.compute.dtu.dk/ahfrom/Formalizing_Axiomatic_Systems_for_Propositional_Logic_in_Isabelle.pdf, 2021.

[8] Julius Michaelis and Tobias Nipkow. Formalized proof systems for propositional logic. In Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, editors, *23rd International Conference on Types for Proofs and Programs, TYPES 2017, May 29-June 1, 2017, Budapest, Hungary*, volume 104 of *LIPIcs*, pages 5:1–5:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[9] Julius Michaelis and Tobias Nipkow. Propositional proof systems. *Archive of Formal Proofs*, June 2017. https://isa-afp.org/entries/Propositional_Proof_Systems.html, Formal proof development.

[10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[11] Frank Pfenning. Single axioms in the implicational propositional calculus. In Ewing L. Lusk and Ross A. Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, volume 310 of *Lecture Notes in Computer Science*, pages 710–713. Springer, 1988.

[12] Larry Wos and Gail W. Pieper. *Automated Reasoning and the Discovery of Missing and Elegant Proofs*. Rinton Press, 2003.

[13] Jan Łukasiewicz. The shortest axiom of the implicational calculus of propositions. *Proceedings of the Royal Irish Academy. Section A: Mathematical and Physical Sciences*, 52:25–33, 1948.

Simulating large eliminations in Cedille

Christopher Jenkins, Andrew Marmaduke, and Aaron Stump

The University of Iowa, Iowa City, Iowa, U.S.A.
{firstname-lastname}@uiowa.edu

1 Introduction

In dependently typed programming languages, large eliminations allow programmers to define types by induction over datatypes — that is, as an elimination of a datatype into the large universe of types. This provides an expressive mechanism for arity- and data-generic programming [7]. However, as large eliminations are closely tied to a type theory's primitive notion of inductive type, this expressivity is not expected within polymorphic pure typed lambda calculi in which datatypes are encoded using impredicative quantification.

Seeking to overcome historical difficulties of impredicative encodings, the *calculus of dependent lambda eliminations* (CDLE) [5, 6] extends the Curry-style (i.e., extrinsically typed) *calculus of constructions* (CC) [1] with three type constructs that together enable the derivation of induction for impredicative encodings of datatypes (Geuvers [3] showed this was not possible for CC). In this paper, we report progress on overcoming another difficulty: the lack of large eliminations for these encodings. We show that the expected computation rules for a large elimination, expressed using a *derivable* notion of extensional equality for types, can be proven within CDLE. We outline our method with a definition of n-ary functions in the remainder of this paper; omitted are many other examples and a generic formulation of the method for the Mendler-style encodings of the framework of Firsov et al. [2]. These results have been mechanically checked by Cedille, an implementation of CDLE.

2 Simulating large eliminations: n-ary functions

Figure 1a shows the definition of *Nary*, the family of n-ary function types over some type T, as a large elimination of natural numbers *Nat*. Our method begins by approximating this inductive definition of a *function* as an inductive *relation* between *Nat* and types, given as *NaryR* in Figure 1b. This approximation is inadequate: we lack a canonical name for the type *Nary n* because n does not *a priori* determine the type argument of *NaryR n*. In fact, without a method of proof discrimination we are unable to define a function of type $\forall\, N.\, \mathit{NaryR}\ \mathit{zero}\ N \to N \to T$ to extract a 0-ary term of type T. In the *naryRS* case, one would need to discharge the absurd equation $\{\mathit{zero} \simeq \mathit{suc}\ n\}$ for some n (the equality type for terms is written $\{t_1 \simeq t_2\}$). CDLE provides such a discriminator with the δ axiom [6] for its primitive equality type, allowing one to abort impossible cases.

(a) As a large elimination

```
Nary : Nat → ⋆
Nary zero = T
Nary (suc n) = T → Nary n
```

(b) As a GADT

```
data NaryR : Nat → ⋆ → ⋆
= naryRZ : NaryR zero T
| naryRS : ∀ n,Y. NaryR n Y → NaryR (suc n) (T → Y)
```

Figure 1: n-ary functions over T

Our task is to show that *NaryR* defines a *functional* relation, i.e., for all $n : Nat$ there exists a unique type *Nary* n such that *NaryR* n (*Nary* n) is inhabited. Using implicit products (c.f. Miquel [4]), a candidate for *Nary* can be defined in CDLE as:

```
Nary = λ n: Nat. ∀ X: ⋆. NaryR n X ⇒ X
```

For all n, read *Nary* n as the type of terms contained in the intersection of the family of types X such that *NaryR* n X is inhabited. For example, every term of type *Nary zero* has type T (since T is in this family), and every term of type T has type *Nary zero* (by induction on the assumed proof of *NaryR zero* X for arbitrary X). However, at the moment we are stuck when attempting to prove *NaryR zero* (*Nary zero*). Though we see that T and *Nary zero* are *extensionally* equal types (they classify the same terms), using *naryRZ* requires that they be *definitionally* equal!

$$\frac{\Gamma \vdash \lambda\, x.\, x : S \to T \quad \Gamma \vdash \lambda\, x.\, x : T \to S}{\Gamma \vdash \lambda\, x.\, x : \{S \cong T\}} \qquad \frac{\Gamma \vdash t : T_j \quad \Gamma \vdash t' : \{T_1 \cong T_2\} \quad i, j \in \{1, 2\}, i \neq j}{\Gamma \vdash t : T_i}$$

Figure 2: Derived extensional equality of types

Figure 2 gives an axiomatic presentation of a derived type family expressing extensional type equality in CDLE, written with curly braces to match the notational convention for equality between terms. The introduction rule states that S and T are equal if the identity function can be assigned both the types $S \to T$ and $T \to S$, i.e., we can exhibit a two-way inclusion between the set of terms of type S and terms of type T. The elimination rule allows us to coerce the type of a term when that type is provably equal to another type. We change the definition of *NaryR* so that its type index respects extensional type equality:

```
data NaryR : Nat → ⋆ → ⋆
= naryRZ : ∀ X. { X ≅ T } → NaryR zero X
| naryRS : ∀ n,Y,X. NaryR n Y → { X ≅ T → Y } → NaryR (suc n) X
```

With the move to an extensional notion of type equality, to show that *NaryR* is functional requires showing that it is *well-defined* with respect to this notion. These three properties — well-definedness, uniqueness, and existence — can be proven in CDLE. We show the types of these proofs below.

```
naryRWd : ∀ n,X1,X2. NaryR n X1 → { X1 ≅ X2 } → NaryR n X2
naryREq : ∀ n,X1,X2. NaryR n X1 → NaryR n X2 → { X1 ≅ X2 }
naryREx : Π n. NaryR n (Nary n)
```

From this, we prove that the computation laws of Figure 1a hold as extensional type equalities:

```
naryZC : { Nary zero ≅ T }
narySC : ∀ n. { Nary (succ n) ≅ T → Nary n }
```

The upshot is we can simulate large eliminations with two-way type inclusions between the left- and right-hand sides of such a definition. For example, the function *app* that applies an n-ary function to a length-indexed list of n elements of type T, written in Agda-like pseudocode as:

```
app : ∀ n. Nary n → Vec T n → T
app .zero f vnil = f
app .(succ n) f (vcons hd tl) = app n (f hd) tl
```

is typeable in CDLE using *naryZC* on f in the case for *vnil* and *narySC* in the case for *vcons*.

References

[1] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[2] Denis Firsov, Richard Blair, and Aaron Stump. Efficient Mendler-style lambda-encodings in Cedille. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252, Cham, 2018. Springer International Publishing.

[3] Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *International Conference on Typed Lambda Calculi and Applications*, pages 166–181, Berlin, Heidelberg, 2001. Springer.

[4] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, page 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.

[5] Aaron Stump. The calculus of dependent lambda eliminations. *J. Funct. Program.*, 27:e14, 2017.

[6] Aaron Stump and Christopher Jenkins. Syntax and semantics of Cedille. Manuscript, 2018.

[7] Stephanie Weirich and Chris Casinghino. Generic programming with dependent types. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 217–258. Springer, 2010.

Quotient inductive-inductive types in the setoid model

Ambrus Kaposi and Zongpu Xie

Eötvös Loránd University, Budapest, Hungary
akaposi@inf.elte.hu and szumixie@gmail.com

Introduction. The setoid model of type theory provides a way to bootstrap functional extensionality [1] and propositional extensionality (univalence for propositions) [3]: the setoid model can be defined in an intensional metatheory with a universe of definitionally proof irrelevant (strict) propositions SProp. Moreover it is a strict model in such a metatheory, that is, all equalities of the model (e.g. β and η for function space) hold definitionally. As a result, we obtain a model construction: any model of type theory with SProp can be turned into another model, its "setoidified" version which supports these extra principles. In addition to functional and propositional extensionality, the setoid model justifies propositional truncation[1], quotient types[2] and countable choice[3].

Since Agda supports SProp [9], it is a convenient tool to experiment with the setoid model. It is straightforward to formalise the setoid model as a category with families (CwF [6]) with Π, Σ, unit, empty, Bool, $\mathbb{N}$, Id types, a universe of strict propositions. Extending the setoid model with a (non-univalent) universe of sets is harder, it was shown by Altenkirch et al. [2] that it can be done using a special form of induction-recursion or large induction-induction (both of which are supported by Agda) or an SProp-valued identity type with transport over types (which is currently not supported by Agda).

Until recently we thought [12] that general inductive types and even quotient inductive-inductive types (QIITs, initial algebras of generalised algebraic theories [13, 5]) are unproblematic in the setoid model, provided we have (possibly SProp-sorted) inductive-inductive types in the metatheory. Simon Boulier pointed out that our formalisations of Martin-Löf's identity type[4] and the universal QIIT[5] only provide eliminators in the empty context. They can be salvaged using a method related to the local universes construction [14] which we explain below.

The setoid model. A context or a closed type in this model is a setoid, i.e. a set (we say set instead of (Agda) type to avoid confusion) together with an SProp-valued equivalence relation. A type over a context $\Gamma = (|\Gamma|, \sim_\Gamma)$ is a displayed setoid with a fibration condition $\mathsf{coe}_A : x \sim_\Gamma x' \to |A|\ x \to |A|\ x'$ such that $x \sim_A (\mathsf{coe}_A\, p\, x)$. Substitutions (and terms) are (dependent) functions between the underlying sets which preserve the relations.

Example: Con-Ty. To illustrate the general method, we explain how to construct the following QIIT in the setoid model[6]. It has two sorts, five constructors and one equality constructor.

$$\begin{array}{ll} \mathsf{Con} & : \mathsf{Set} \\ \mathsf{Ty} & : \mathsf{Con} \to \mathsf{Set} \\ \bullet & : \mathsf{Con} \\ - \triangleright - & : (\gamma : \mathsf{Con}) \to \mathsf{Ty}\ \gamma \to \mathsf{Con} \end{array} \qquad \begin{array}{ll} \mathsf{U} & : \mathsf{Ty}\ \gamma \\ \mathsf{El} & : \mathsf{Ty}\ (\gamma \triangleright \mathsf{U}) \\ \Sigma & : (a : \mathsf{Ty}\ \gamma) \to \mathsf{Ty}\ (\gamma \triangleright a) \to \mathsf{Ty}\ \gamma \\ \mathsf{eq} & : \gamma \triangleright \Sigma\ a\ b = \gamma \triangleright a \triangleright b \end{array}$$

[1] https://bitbucket.org/akaposi/setoid/src/master/agda/Model/Trunc.agda
[2] https://bitbucket.org/akaposi/setoid/src/master/agda/Model/Quotient.agda
[3] https://bitbucket.org/akaposi/setoid/src/master/agda/Model/CountableChoice.agda
[4] https://bitbucket.org/akaposi/qiit/src/master/Setoid/Path.agda
[5] https://bitbucket.org/akaposi/qiit/src/master/Setoid/UniversalQIIT/
[6] https://bitbucket.org/akaposi/qiit/src/master/Setoid/ConTy2.agda

We omitted some arguments, e.g. U implicitly takes a parameter γ. In the setoid model, we need to define a type Con in the empty context, a type Ty over Con, and their elimination principles. We start by defining in Agda an inductive-inductive type (IIT) with these sorts:

$$|\mathsf{Con}| : \mathsf{Set} \qquad |\mathsf{Ty}| : |\mathsf{Con}| \to \mathsf{Set}$$
$$\sim_{\mathsf{Con}} : |\mathsf{Con}| \to |\mathsf{Con}| \to \mathsf{SProp} \qquad \sim_{\mathsf{Ty}} : \gamma \sim_{\mathsf{Con}} \gamma' \to |\mathsf{Ty}|\,\gamma \to |\mathsf{Ty}|\,\gamma' \to \mathsf{SProp}$$

The constructors of $|\mathsf{Con}|$ are $|\bullet|$ and $|\triangleright|$, the constructors of $|\mathsf{Ty}|$ are $|\mathsf{U}|$, $|\mathsf{El}|$ and $|\Sigma|$, while $\sim_{\mathsf{Con}}$ has a constuctor $|\mathsf{eq}|$. In addition, $\sim_{\mathsf{Con}}$ and $\sim_{\mathsf{Ty}}$ have constructors stating that it is an equivalence relation, and they have congruence constructors for each point constructor, e.g. there is $\sim_{\triangleright}$: $(p : \gamma \sim_{\mathsf{Con}} \gamma') \to \sim_{\mathsf{Ty}} p\, a\, a' \to (\gamma \triangleright a) \sim_{\mathsf{Con}} (\gamma' \triangleright a')$. Finally, $|\mathsf{Ty}|$ has a constructor $\mathsf{coe}_{\mathsf{Ty}} : \gamma_0 \sim_{\mathsf{Con}} \gamma_1 \to |\mathsf{Ty}|\,\gamma_0 \to |\mathsf{Ty}|\,\gamma_1$ and $\sim_{\mathsf{Ty}}$ a constructor for $\sim_{\mathsf{Ty}} p\, a\, (\mathsf{coe}_{\mathsf{Ty}}\, p\, a)$. Thus the IIT is the "fibrant equivalence congruence closure" of the constructors.

With the aid of this IIT (note that it has both Set and SProp-sorts) we define the type formation rules and constructors of the Con-Ty QIIT in the setoid model *in the empty context*: the underlying set for Con is $|\mathsf{Con}|$, the relation is $\sim_{\mathsf{Con}}$, the witnesses for the equivalence relation come from the corresponding constructors of $\sim_{\mathsf{Con}}$, and so on. Thus Con becomes a type in the empty context in the setoid model. Ty is a type over the one-element context Con. $\bullet$ is a term in the empty context of type Con, and so on. We added exactly the required structure to the IIT to be able to define the constructors. The eq equality constructor is given by $|\mathsf{eq}|$. Given a Con-Ty algebra in the empty context, we define four functions by recursion-recursion as a first step towards the (non-dependent) elimination principle.

The type formation rules and constructors can easily be lifted from the empty context to an arbitrary context and all the substitution laws hold definitionally. We also need that for any context Γ, we can eliminate into a Con-Ty algebra in Γ. Our setoid model has Π types and K constant types (a context can be turned into a type). With the help of these we can turn a type C in Γ into the type $\Pi(x : \mathsf{K}\,\Gamma).C[x]$ which is in the empty context. This way we turn the algebra in Γ into an algebra in the empty context on which we can apply our previously defined elimination principle. This way we obtain the eliminator in arbitrary contexts. All computation rules of this eliminator are definitional.

We prove uniqueness of the eliminator by induction-induction on $|\mathsf{Con}|$ and $|\mathsf{Ty}|$. The substitution law of the eliminator is proven by another induction-induction on the same sets.

In our formalisation, Con-Ty has an additional infinitary constructor (an infinitary Π type indexed by a code of a setoid in a universe). It seems that with the help of a universe in the setoid model, open QIITs and those with infinitary constructors can be handled as well. Note that in contrast with the unordered infinitely branching tree example in [4], we do not use the (countable) axiom of choice to construct this QIIT.

Arbitrary QIITs Signatures for QIITs can be specified using the theory of QIIT signatures [13] (ToS) which is itself an infinitary QIIT. We formalised[7] that the setoid model supports ToS in the empty context. Based on our experience with Con-Ty, we expect that it is possible to lift ToS from the empty context to arbitrary contexts. If this succeeds, then following [13], we can construct all QIITs from ToS with propositional computation rules. As the construction of [13] is performed in extensional type theory, we use Hofmann's conservativity result [10, 15] to transfer it to the setoid model. This way however we only obtain propositional computation rules. It remains to be proven that all QIITs are supported by the setoid model with definitional computation rules. We plan to do this by induction on QIIT signatures.

We would also like to understand the relationship of this construction to that of higher inductive types (HITs) in cubical models [8, 7] and how they could be extended to HIITs [11].

[7] https://bitbucket.org/akaposi/qiit/src/master/Setoid2/ToS/

References

[1] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[2] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, Christian Sattler, and Filippo Sestini. Constructing a universe for the setoid model. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12650 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2021.

[3] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory—a syntactic translation. In Graham Hutton, editor, *Mathematics of Program Construction*, pages 155–196, Cham, 2019. Springer International Publishing.

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

[5] John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[6] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019.

[7] Evan Cavallo and Robert Harper. Higher inductive types in cubical computational type theory. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[8] Thierry Coquand, Simon Huber, and Anders Mörtberg. On higher inductive types in cubical type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 255–264, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, pages 1–28, January 2019.

[10] Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *TYPES 95*, pages 153–164, 1995.

[11] Ambrus Kaposi and András Kovács. Signatures and Induction Principles for Higher Inductive-Inductive Types. *Logical Methods in Computer Science*, Volume 16, Issue 1, February 2020.

[12] Ambrus Kaposi and Zongpu Xie. A model of type theory with quotient inductive-inductive types. In Ugo de' Liguoro and Stefano Berardi, editors, *26th International Conference on Types for Proofs and Programs, TYPES 2020*. University of Turin, 2020.

[13] András Kovács and Ambrus Kaposi. Large and infinitary quotient inductive-inductive types. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 648–661. ACM, 2020.

[14] Peter Lefanu Lumsdaine and Michael A. Warren. The local universes model: An overlooked coherence construction for dependent type theories. *ACM Trans. Comput. Logic*, 16(3), July 2015.

[15] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. Eliminating reflection from type theory. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 91–103. ACM, 2019.

Using Two-Level Type Theory for Staged Compilation *

András Kovács

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

Two-level type theory [2] (2LTT) is a system for performing certain metatheoretic constructions and reasoning involving object-level type theories. Such reasoning is always possible by simply embedding object theories in metatheories, but in that case we have to explicitly handle a deluge of technical details about the object theory, most notably substitutions. If everything that we aim to do is natural with respect to object-level substitution, we can instead use 2LTT, which can be viewed as a notation for working with presheaves over the object-level category of substitutions.

Likewise in metaprogramming, there is a spectrum: we can simply write programs which output raw source code, or use staging instead, which is safer and more convenient, but also restricted in some ways. In the current work we observe that 2LTT is a powerful model for generative staged compilation.

Basic rules of 2LTT. We have universes U^s_i, where $s \in \{0, 1\}$, denoting a *stage* or level in the 2LTT sense, and $i \in \mathbb{N}$ denotes a usual level index of *sizing* hierarchies. The two dimensions of indexing are orthogonal, and we will elide the i indices in the following. We assume Russell-style universes. For each $\Gamma \vdash A : \mathsf{U}^0$, we have $\Gamma \vdash \mathsf{Code}\, A : \mathsf{U}^1$. Quoting: for each $\Gamma \vdash t : A$, we have $\Gamma \vdash <t> : \mathsf{Code}\, A$. Unquoting: for each $\Gamma \vdash t : \mathsf{Code}\, A$, we have $\Gamma \vdash \sim t : A$. Moreover, quoting and unquoting form an isomorphism up to definitional equality. U^0 and U^1 can be closed under arbitrary additional type formers.

The idea of staging is the following: given a closed $A : \mathsf{U}^0$ with a closed $t : A$ in 2LTT, there are unique A' and t' in the object theory, which become definitionally equal to A and t respectively after being embedded in 2LTT. In short, every meta-level construction can be computed away, and only object-level constructions remain in the result. Annekov et al. [2] only showed mere existence of A' and t' (as a conservativity theorem for 2LTT). We can get unique existence as well, using the normalization of 2LTT: by induction on the (unique) normal forms of A and t, we can show that they cannot contain meta-level subterms. This shows that normalization is a sound staging algorithm, but in practice we do not want to compute full normal forms; we want to compute *meta-level redexes only*. This can be done with a variation of standard normalization-by-evaluation [1, 5] which also keeps track of stages.

Applications

Control over inlining and compile-time computation. We can define two variations of the polymorphic identity function for object-level types:

$$\begin{aligned} &id : (A : \mathsf{U}^0) \to A \to A && id' : (A : \mathsf{Code}\, \mathsf{U}^0) \to \mathsf{Code}\ {\sim}A \to \mathsf{Code}\ {\sim}A \\ &id = \lambda\, A\, x.\, x && id' = \lambda A\, x.x \end{aligned}$$

The second version is evaluated at compile time. For example, $\sim(id' < \mathsf{Bool}^0 > < \mathsf{true}^0 >)$ can be used in object-level code, which is computed to true^0 by staging. We can also freely

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

use induction on meta-level values to generate object-level code, including types. Hence, 2LTT supports full dependent types (with universes and large elimination) in staging.

Monomorphization. We assume now that the object language is a *simple type theory*. In this case, there is no universe U^0 in the object-level, so there is no $\mathsf{Code}\,\mathsf{U}^0$, but we can still freely include a meta-level type Ty^0 whose terms are identified with object-level types. Now, meta-level functions can be used for quantification over object-level types, as in $id : (A : \mathsf{Ty}^0) \to \mathsf{Code}\,A \to \mathsf{Code}\,A$. However, since the object theory is simply typed and monomorphic, all polymorphism is guaranteed to compute away during staging.

Control over lambda lifting and closure creation. We assume now a dependent type theory on both levels, but with a *first-order function type* on the object level. This is defined by splitting U^0 to a universe V^0 which is closed under inductive types but not functions, and a universe C^0 which has V^0 as a sub-universe, and is closed under functions with domains in V^0 and codomains in C^0. This object theory supports compilation which requires only lambda lifting, but no closures. On its own, the object theory is fairly restricted, but together with staging we have a remarkably expressive system. Then, we can close V^0 under a separate type former of closure-based functions, thereby formally distinguishing lambda-liftable functions from closure-based functions. This enables typed analysis of various optimization and fusion techniques. E.g. we get guaranteed closure-freedom in code output if a certain fusion technique can be formalized with only first-order function types. In particular, this may obviate the need for arity analysis [3] in fold-based fusion.

Memory layout control. We assume again a dependent theory on both levels, but now index U^0 over *memory layouts*. For example, $\mathsf{U}^0\,\mathsf{erased}$ may contain runtime-erased types, and $\mathsf{U}^0\,(\mathsf{word64} \times \mathsf{word64})$ may contain types represented as unboxed pairs of machine words. We assume a meta-level type of layouts. Hence, we can abstract over layouts, but after staging every layout will be concrete and canonical in the output. This can be viewed as a more powerful version of *levity polymorphism* in GHC [4], and a way to retain both dependent types and non-uniform memory layouts in the object theory.

Potential extensions

More stages, stage polymorphism. The standard presheaf semantics of 2LTT can be extended to more levels in a straightforward way. It seems feasible to also allow quantifying over all smaller levels, at a given level.

Stage inference. Code preserves all negative type formers up to definitional isomorphism [2], e.g. $\mathsf{Code}\,(A \to B) \simeq (\mathsf{Code}\,A \to \mathsf{Code}\,B)$. This can be used to support inference for staging annotations, by automatically inserting transports along preservation isomorphisms during elaboration. The previous $\sim(id' < \mathsf{Bool}^0 > < \mathsf{true}^0 >)$ example could be simply written as $id'\,\mathsf{Bool}^0\,\mathsf{true}^0$ in the surface language, and elaboration would transport id' appropriately. We demonstrated the practical feasibility of such stage inference in a prototype implementation.

Induction on Code. Basic 2LTT supports *any model* of the object theory in the presheaf semantics, it does not assume that we have presheaves over the initial model (syntax). Hence, $\mathsf{Code}\,A$ is a black box without elimination principles. However, if we are interested in staged compilation, we can assume the object level to be syntactic and consistently add operations on $\mathsf{Code}\,A$ which rely on that assumption, e.g. conversion checking, pattern matching, or induction on normal forms of object-level expressions.

References

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

[2] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *ArXiv e-prints*, may 2019.

[3] Joachim Breitner. Call arity. *Comput. Lang. Syst. Struct.*, 52:65–91, 2018.

[4] Richard A. Eisenberg and Simon Peyton Jones. Levity polymorphism. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 525–539. ACM, 2017.

[5] PawełWieczorek and Dariusz Biernacki. A Coq formalization of normalization by evaluation for Martin-Löf type theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 266–279, New York, NY, USA, 2018. ACM.

Constructive Notions of Ordinals in Homotopy Type Theory*

Nicolai Kraus[1], Fredrik Nordvall Forsberg[2], and Chuangjie Xu[3]

[1] University of Nottingham, Nottingham, UK
[2] University of Strathclyde, Glasgow, UK
[3] fortiss GmbH, Munich, Germany

Introduction Ordinals are numbers that, although possibly infinite, share an important property with the natural numbers: every decreasing sequence necessarily terminates. This makes them a powerful tool when proving that processes terminate, or justifying induction and recursion [DM79, Flo67]. There is also a rich theory of arithmetic on ordinals, generalising the usual theory of arithmetic on the natural numbers. Unfortunately, the standard definition of ordinals is not very well-behaved constructively, and the notion fragments into a number of inequivalent definitions, each with pros and cons. We consider three different constructive notions in homotopy type theory, and show how they relate to each other.

Cantor Normal Forms as a Subset of Binary Trees In classical set theory, it is well known that every ordinal α can be written uniquely in Cantor normal form

$$\alpha = \omega^{\beta_1} + \omega^{\beta_2} + \cdots + \omega^{\beta_n} \text{ with } \beta_1 \geq \beta_2 \geq \cdots \geq \beta_n \tag{1}$$

for some natural number n and ordinals β_i. If $\alpha < \varepsilon_0$, then $\beta_i < \alpha$, and we can represent α as a finite binary tree (with a condition) as follows [Buc91, NXG20]. Let $\mathcal{T}$ be the type of unlabeled binary trees, i.e. the inductive type with suggestively named constructors $0 : \mathcal{T}$ and $\omega^- \mathbin{+} - : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$. Let the relation $<$ be the lexicographical order, i.e. generated by the following clauses:

$$0 < \omega^a \mathbin{+} b \qquad a < c \to \omega^a \mathbin{+} b < \omega^c \mathbin{+} d \qquad b < d \to \omega^a \mathbin{+} b < \omega^a \mathbin{+} d.$$

We have the map $\mathsf{left} : \mathcal{T} \to \mathcal{T}$ defined by $\mathsf{left}(0) :\equiv 0$ and $\mathsf{left}(\omega^a \mathbin{+} b) :\equiv a$ which gives us the left subtree (if it exists) of a tree. A tree is a *Cantor normal form* (CNF) if, for every $\omega^s \mathbin{+} t$ that the tree contains, we have $\mathsf{left}(t) \leq s$, where $s \leq t :\equiv (s < t) \uplus (s = t)$; this enforces the condition in (1). Formally, the predicate isCNF is defined inductively by the two clauses

$$\mathsf{isCNF}(0) \qquad \mathsf{isCNF}(s) \to \mathsf{isCNF}(t) \to \mathsf{left}(t) \leq s \to \mathsf{isCNF}(\omega^s \mathbin{+} t).$$

We write $\mathsf{Cnf} :\equiv \Sigma(t : \mathcal{T}).\mathsf{isCNF}(t)$ for the type of Cantor normal forms.

Brouwer Trees as a Quotient Inductive-Inductive Type In the functional programming community, it is popular to consider *Brouwer ordinal trees* $\mathcal{O}$ as inductively generated by zero, successor and a "supremum" constructor $\mathsf{sup} : (\mathbb{N} \to \mathcal{O}) \to \mathcal{O}$ which forms a new tree for every countable sequence of trees [Bro26, CHS97, Han00]. By the inductive nature of the definition, constructions on trees can be carried out by giving one case for zero, one for successors, and one for suprema, just as in the classical theorem of transfinite induction. However, calling the constructor sup is wishful thinking; $\mathsf{sup}(s)$ does not faithfully represent the suprema of

*Supported by the Royal Society, grant reference URF\R1\191055, the UK National Physical Laboratory Measurement Fellowship project *Dependent types for trustworthy tools*, and the LMUexcellent program.

the sequence s, since we do not have that e.g. $\mathsf{sup}(s_0, s_1, s_2, \ldots) = \mathsf{sup}(s_1, s_0, s_2, \ldots)$ — each sequence gives rise to a new tree, rather than identifying trees representing the same supremum.

Using a *quotient inductive-inductive type* [ACD+18], we can remedy the situation: Let A be a type and $\prec : A \to A \to \mathsf{hProp}$. For sequences $f, g : \mathbb{N} \to A$, we say that f is *simulated by* g if $f \precsim g :\equiv \forall k.\exists n.f(k) \prec g(n)$ (where $\exists$ is truncated Σ). We say that f and g are *bisimilar* with respect to $\prec$, written $f \approx^{\prec} g$, if we have both $f \precsim g$ and $g \precsim f$. A sequence $f : \mathbb{N} \to A$ is *increasing* with respect to $\prec$ if we have $\forall k.f(k) \prec f(k+1)$. We write $\mathbb{N} \xrightarrow{\prec} A$ for the type of $\prec$-increasing sequences. We now mutually construct the type $\mathsf{Brw} : \mathsf{hSet}$ together with a relation $\leq : \mathsf{Brw} \to \mathsf{Brw} \to \mathsf{hProp}$. The constructors for Brw are $\mathsf{zero} : \mathsf{Brw}$, $\mathsf{succ} : \mathsf{Brw} \to \mathsf{Brw}$, and

$$\mathsf{limit} : (\mathbb{N} \xrightarrow{<} \mathsf{Brw}) \to \mathsf{Brw} \qquad \text{and} \qquad \mathsf{bisim} : f \approx^{\leq} g \to \mathsf{limit}\, f = \mathsf{limit}\, g,$$

where we denote $x < y :\equiv \mathsf{succ}\, x \leq y$ in the type of limit. The constructors for $\leq$ ensure transitivity, that zero is minimal, that succ is monotone, and that $\mathsf{limit}\, f$ is the least upper bound of f. Because of the infinitary constructor limit, we lose full decidability of equality and order relations, but by restricting to limits of increasing sequences, we retain the possibility of classifying an ordinal as zero, a successor, or a limit.

Extensional Wellfounded Orders Finally, we consider a variation on the classical set-theoretical axioms for ordinals more suitable for a constructive treatment [Tay96], following the HoTT book [Uni13, Chapter 10] and Escardó [Esc21]. The type Ord consists of a type X together with a relation $\prec : X \to X \to \mathsf{hProp}$ which is *transitive*, *extensional* (any two elements of with the same predecessors are equal), and *wellfounded* (every element is accessible, where accessibility is the least relation such that x is accessible if every $y \prec x$ is accessible.).

We also have a relation on Ord itself. Following [Uni13, Def 10.3.11 and Cor 10.3.13], a *simulation* between ordinals $(X, \prec_X)$ and $(Y, \prec_Y)$ is a monotone function $f : X \to Y$ such that for all $x : X$ and $y : Y$, if $y \prec_Y f\, x$, then we have an $x_0 \prec_X x$ such that $f\, x_0 = y$. We write $X \leq Y$ for the type of simulations between $(X, \prec_X)$ and $(Y, \prec_Y)$. Given an ordinal $(X, \prec)$ and $x : X$, the *initial segment* of elements below x is given as $X_{/x} :\equiv \Sigma(y : X).y \prec x$. A simulation $f : X \leq Y$ is *bounded* if we have $y : Y$ such that f induces an equivalence $X \simeq Y_{/y}$. We write $X < Y$ for the type of bounded simulations.

Results For each of Cnf, Brw, Ord, the relation $<$ is transitive, extensional, and wellfounded; for wellfoundedness, the refined definitions of Cnf and Brw which excludes "junk" terms are crucial. For Cnf, $<$ is decidable, whereas for Ord, $<$ is decidable if and only if the law of excluded middle holds. Brw sit in the middle, with some of its properties being decidable, e.g. it is decidable whether a given x is finite, but $<$ is not decidable in general without further assumptions. We introduce an abstract framework axiomatising properties such as being a successor or a limit ordinal, which makes it possible to compare the different notions of ordinals above. According to these definitions, each of Cnf, Brw, Ord has zeroes and successors, and the successor functions of Cnf and Brw are both $<$- and $\leq$-monotone. For the successor function of Ord, each of the two monotonicity properties on its own is equivalent to the law of excluded middle. Cnf does not have limits, but both Brw and Ord do. Using the abstract notions of zero, successor and limit, we can give an abstract specification of the arithmetic operations; we say that a notion of ordinals has unique arithmetic if the type of implementations of the specification is contractible. Cnf has addition, multiplication, and exponentiation with base ω (all unique), Brw has addition, multiplication and exponentiation with every base (all unique), and Ord has addition and multiplication. Finally, we have order-preserving embeddings $\mathsf{Cnf} \hookrightarrow \mathsf{Brw} \hookrightarrow \mathsf{Ord}$.

Details and Formalisation Full details: arxiv:2104.02549. We have formalised our results in cubical Agda: https://bitbucket.org/nicolaikraus/constructive-ordinals-in-hott.

References

[ACD+18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *FoSSaCS '18*, pages 293–310. Springer, 2018.

[Bro26] L. E. J. Brouwer. Zur begründung der intuitionistische mathematik III. *Mathematische Annalen*, 96:451–488, 1926.

[Buc91] Wilfried Buchholz. Notation systems for infinitary derivations. *Archive for Mathematical Logic*, 30:227–296, 1991.

[CHS97] Thierry Coquand, Peter Hancock, and Anton Setzer. Ordinals in type theory. Invited talk at Computer Science Logic (CSL), http://www.cse.chalmers.se/~coquand/ordinal.ps, 1997.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[Esc21] Martín Escardó. Agda implementation: Ordinals, Since 2010–2021. https://www.cs.bham.ac.uk/~mhe/TypeTopology/Ordinals.html.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967.

[Han00] Peter Hancock. *Ordinals and Interactive Programs*. PhD thesis, University of Edinburgh, 2000.

[NXG20] Fredrik Nordvall Forsberg, Chuangjie Xu, and Neil Ghani. Three equivalent ordinal notation systems in cubical Agda. In *CPP '20*, pages 172–185. ACM, 2020.

[Tay96] Paul Taylor. Intuitionistic sets and ordinals. *Journal of Symbolic Logic*, 61(3):705–744, 1996.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book/, Institute for Advanced Study, 2013.

Family Polymorphism for Proof Extensibility

Anastasiya Kravchuk-Kirilyuk[1], Yizhou Zhang[2], and Nada Amin[1]

[1] Harvard University, Cambridge, Massachusetts, U.S.A.
akravchukkirilyuk@g.harvard.edu, namin@seas.harvard.edu
[2] University of Waterloo, Waterloo, Ontario, Canada
yizhou@uwaterloo.ca

Abstract

Proof assistants provide users with a wide variety of tactics, but hardly any clear mechanisms for modular proof extensibility. Oftentimes, entire developments are copied with minor changes, and connected developments grow apart due to untracked changes. We aim to solve this problem by implementing a modular, family-polymorphic system for proof extensibility in Coq. As opposed to most existing work, our solution will be an extension to Coq theory (the Calculus of Inductive Constructions), and will be made accessible to the user via a Coq plug-in.

1 Introduction

Theorem provers such as the Coq Proof Assistant [1] allow us to design and develop mathematical models and write proofs about them. While most proof assistants offer a wide pool of proof tactics, extensibility of existing proofs is not usually a built-in feature. Any changes to a model's definitions, or simply adding constructors to an inductive data type, can require a lengthy propagation of those changes through existing lemmas and proofs. Even worse, the "old" development is often copied and pasted with minor changes. This creates a lot of duplicated code and unintentionally disjoint developments. Any further changes made in the "old" development will have to be manually implemented in the "new" development. In another scenario, we may have an abstract base framework which can be instantiated in different ways to produce derived frameworks. Ideally, we should be able to re-use proofs about the base framework in these instances, as opposed to verifying every derived framework from scratch. We hope that our family polymorphic framework can solve such extensibility problems and ultimately inspire built-in extensibility support in proof assistants.

2 Background on Family Polymorphism

The concept of *family polymorphism* was introduced by Ernst in 2001 [6], as a way to achieve safety and flexibility in multi-object systems. This is best illustrated by a method call example. Consider a single-object scenario: object **x** (an instance of class **X**) invokes some method **m(...)**. In this case, we need only make sure that the method definition for **m** is compatible with class **X**. However, when another object instance **y** of class **Y** is added to the mix (e.g., as an argument to **m**), we have to ensure the compatibility of **m** with respect to both classes **X** and **Y**. Even worse, instances of **X** and **Y** can be instances of any subclasses of **X** and **Y**, and not all subclasses may be compatible with *each other*. Having many separate definitions for **m** is not a flexible solution. Instead, to ensure flexibility and safety, compatible classes can be grouped into *families*. Classes within a *family* can safely interact with each other. Conversely, members of different families are incompatible and should not interact. Then, through the use of relative path types and late binding we can guarantee that only instances of classes *from the same family* interact.

3 Approach

In a similar fashion, we can imagine *families* of definitions and proofs. From any *base family* we can create any number of *derived families* through inheritance. All the definitions and proofs in the base family can be extended to yield the respective constructs for the derived family. Constructs from different derived families will not interact with each other. The two-fold advantage of family polymorphism still stands: the flexibility of extensible constructs is combined with the guarantee of safe interaction within families.

To apply this approach to Coq, we must first translate the object-oriented notion of family polymorphism to the functional, and then dependently typed, setting. We are currently building a modular, family-polymorphic system which can adapt to a functional setting with records and (G)ADTs. Our approach is inspired by the class-based family polymorphism of Igarashi et al. [8]. In our system, *families* are top-level structures which contain *type definitions* (represented by records) and *function definitions* (represented by lambda abstractions). We make use of *relative path types* to ensure all interactions are within a family, like Ernst [6]. We also track inherited family members through the use of *linkages*, in a similar fashion to Zhang et al. [13]. A *linkage* allows us to concisely keep track of the entire family inheritance tree: newly defined and extended types, newly defined functions, and re-defined functions with identical signatures. A family *linkage* makes it easy for us to retrieve the proper (and compatible!) type or function definition.

4 Related Work on Extensibility of Coq

Typeclasses in Coq [10] achieve extensibility through genericity, but cannot achieve the extensions we have in mind (e.g., a model augmented by a case in an algebraic data type). Coq à la carte [7] and Meta-theory à la carte [3, 4, 5] solve the expression problem [12] by encodings and design patterns, without changing the Coq theory. We aim instead for an extension to the Coq theory so that we can avoid verbosity. Relevant Coq plug-ins include Pumpkin Pi [9] and Coq-Elpi [11]. The former repairs proofs over changed types (for changes represented as equivalences) [9], while the latter allows for extending Coq with new commands and tactics, and manipulating Coq terms with binders [11]. One way to achieve extensible pattern matching – which is relevant to our extensibility goals – is via the use of first-class cases [2].

5 Future Work

Next, we will pinpoint desired extensibility features and design a set of strategies for modular reuse in Coq. We will consider both extensibility of definitions and extensibility of proofs. We should handle extensibility of existing definitions (adding cases to inductive data types or inductive propositions) as well as creating new definitions. We should also achieve consistent and flexible proof extension to accommodate the changed definitions. We should aim to reuse existing proofs with only minor additions (e.g., new proof subgoals for new cases). We should ensure extensibility of pattern matching and ensure that matches in a derived family are exhaustive (consider all new cases). When pattern matching in the base family, we must accommodate future extensions. Although this can be done via a wildcard case in the base module, we may not want such a catch-all in a derived module. We envision the handling of extra proof and pattern match cases via Coq obligations. We will also consider more involved extensions such as adding fields to a constructor, monadic return types, and changes in proof structure.

References

[1] The Coq Proof Assistant. https://coq.inria.fr/.

[2] Matthias Blume, Umut A Acar, and Wonseok Chae. Extensible programming with first-class cases. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 239–250, 2006.

[3] Benjamin Delaware, William Cook, and Don Batory. Product lines of theorems. *ACM SIGPLAN Notices*, 46(10):595–608, oct 2011.

[4] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. *ACM SIGPLAN Notices*, 48(1):207–218, jan 2013.

[5] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming - ICFP '13*, page 319, New York, New York, USA, sep 2013. ACM Press.

[6] Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming*, pages 303–326. Springer, 2001.

[7] Yannick Forster and Kathrin Stark. Coq à la carte: a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 186–200, New York, NY, USA, jan 2020. ACM.

[8] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In Kwangkeun Yi, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, and Gerhard Weikum, editors, *Programming languages and systems*, volume 3780 of *Lecture notes in computer science*, pages 161–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[9] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof repair across type equivalences. *PLDI*, 2021.

[10] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture notes in computer science*, pages 278–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[11] Enrico Tassi. Elpi: an extension language for coq. In *Fourth International Workshop on Coq for Programming Languages*, 2018.

[12] Philip Wadler et al. The expression problem,. Discussion on Java-Genericity mailing list, November 1998. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[13] Yizhou Zhang and Andrew C Myers. Familia: unifying interfaces, type classes, and family polymorphism. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–31, 2017.

A proof assistant for synthetic ∞-categories*

Nikolai Kudasov

Innopolis University, Innopolis, Republic of Tatarstan, Russia
n.kudasov@innopolis.ru

Since the introduction of homotopy type theory [10] many implementations have emerged [12, 11]. Recently, various directed type theories have been developed [4, 7], some of which build up on homotopy type theory and also do not have a proper proof assistant support. We report a work in progress on the implementation[1] of a proof assistant for synthetic ∞-categories based on the type theory developed by Riehl and Shulman [7]. We highlight two of the challenges: (i) interleaving computation with typechecking, and (ii) higher-order unification. The former makes it hard to apply some common approaches to implementation of dependently typed languages, while the latter is typically omitted in prototypes. We suggest a general approach based on free monads and generalized de Bruijn indices that allows to address both problems.

Type theory for synthetic ∞-categories. Riehl and Shulman have introduced a type theory with shapes [7] and, using a directed interval shape $\mathbb{2}$, were able to develop a synthetic theory of $(\infty, 1)$-categories within that type theory. Extension types [7, Figure 4] are central for the type theory (e.g $\hom_A(x, y)$ is an extension type) and allow to reduce bookkeeping in proofs due to the use of judgemental equality. However, that also makes them challenging to implement in a proof assistant. Consider the following rule:

$$\frac{\{t : I \mid \phi\}\,\mathsf{shape} \quad \{t : I \mid \psi\}\,\mathsf{shape} \quad t : I \mid \phi \vdash \psi \qquad \Xi \mid \Phi \mid \Gamma \vdash f : \left\langle \prod_{t:I\mid\psi} A \middle| \begin{smallmatrix}\phi\\ a\end{smallmatrix} \right\rangle \quad \Xi \vdash s : I \quad \Xi \mid \Phi \vdash \phi[s/t]}{\Xi \mid \Phi \mid \Gamma \vdash f(s) \equiv a[s/t]}$$

According to this rule to reduce $f(s)$ it is sometimes enough to know the type of f and that $\phi[s/t]$ holds, making computation depend on type information. This rule is featured in several proofs of [7, Section 4], specifically it is used to verify judgemental equalities imposed by extension types. For example, the following is an analogue of equivalence $X \to (Y \to Z) \simeq Y \to (X \to Z)$, but for dependent functions and extension types:

Theorem 1. *[7, Theorem 4.1] If $t : I \mid \phi \vdash \psi$ and $X : U$, while $Y : \{t : I \mid \psi\} \to X \to U$ and $f : \prod_{t:I\mid\phi} \Phi_{x:X} Y(t, x)$, then*

$$\left\langle \prod_{t:I\mid\psi} \left(\prod_{x:X} Y(t,x) \right) \middle| \begin{smallmatrix}\phi\\ f\end{smallmatrix} \right\rangle \simeq \prod_{x:X} \left\langle \prod_{t:I\mid\psi} Y(t,x) \middle| \begin{smallmatrix}\phi\\ \lambda t.f(t,x)\end{smallmatrix} \right\rangle$$

Partial proof. From right to left we have $h \mapsto \lambda t.\lambda x.h(x, t)$. For this to typecheck we need to verify that $\lambda t.\lambda x.h(x, t) \equiv f$, assuming ϕ. From the type of h and assuming ϕ we can deduce that $h(x) \equiv \lambda t.f(t, x)$. From there we get $\lambda t.\lambda x.h(x, t) \equiv \lambda t.\lambda x.f(t, x) \equiv f$ assuming ϕ. □

In our prototype type checker, the statement and proof of Theorem 1 look like this:

*Other people who contributed to this document include Benedikt Ahrens and Daniel de Carvalho.

[1] work in progress implementation can be found in a GitHub repository at https://github.com/fizruk/rzk

```
Theorem-4.1-right-to-left
  : (I : CUBE) -> (psi : (t : I) -> TOPE) -> (phi : {(t : I) | psi t} -> TOPE)
  -> (X : U) -> (Y : <{t : I | psi t} -> (x : X) -> U >)
  -> (f : <{t : I | phi t} -> (x : X) -> Y t x >)
  -> ((x : X) -> <{t : I | psi t} -> Y t x [phi t |-> f t x]>)
  -> <{t : I | psi t} -> (x : X) -> Y t x [phi t |-> f t]>
  := \I -> \psi -> \phi -> \X -> \Y -> \f -> \h -> \t -> \x -> (h x) t
```

Here the user relies entirely on the proof assistant to typecheck this definition. The typechecker has to apply the aforementioned computation rule which in turn relies on the type information. This situation forbids a simple approach of evaluating terms to values before typechecking as is sometimes done for dependently typed languages [5], if we do not want to litter proofs with explicit type annotations.

Higher-order unification. Unification plays an important role in typechecking as well as type and term inference. In presence of dependent types, some form of higher-order unification is typically expected from the proof assistant to reduce explicit types in proofs. Even though many higher-order unification algorithms exist [3, 6, 8], when prototyping a proof assistant an extra effort is required to add this feature and, being non-trivial, it is often omitted, opting out for a simpler implementation while limiting capabilities of a prototype. However, in a sufficiently complex dependently typed language even small examples can be difficult to comprehend without some type inference. For example, in the presence of type inference with higher-order unification an earlier example could be simplified to something like this:

```
Theorem-4.1-right-to-left I psi phi X Y f
  : ((x : X) -> <{t : I | psi t} -> Y t x [phi t |-> f t x]>)
  -> <{t : I | psi t} -> (x : X) -> Y t x [phi t |-> f t]>
  := \h -> \t -> \x -> (h x) t
```

Free scoped monads. Free monads is one of the popular approaches to modelling side effects (such as input/output) in embedded domain specific languages [13, 14]. Sometimes they are also used to generate the type of expression trees [9], where binding operation corresponds to substitution. When representing expressions, free monads provide flexibility, both in terms of modular extensions [9] and annotations (such as source code location or type information).

For expressions with scopes (such as let-expressions or lambda abstractions) substitution (implemented manually or via free monads) is not safe by default, meaning that a name capture might happen. To avoid name capture de Bruijn indices [2] are commonly used. However, generalized de Bruijn indices[2] have also been used to keep track of scoping in types and also to allow lifting entire subexpressions to further optimize substitution.

We propose a combination of these two techniques (free monads and generalized de Bruijn indices) to simplify implementation of languages[3], in particular dependently typed ones. Our approach can be compared with the recent work on the type and scope safe universe of syntaxes with binding [1]. We also show that it is possible to implement a generic higher-order unification algorithm, based on Huet's algorithm [3] for languages defined using our approach. We present an implementation of this generic algorithm together with example definitions for languages from untyped lambda calculus to basic homotopy type theory without higher inductive types using the Haskell programming language.

[2] such as implemented in the `bound` package, available at http://hackage.haskell.org/package/bound

[3] or, at the very least, prototyping of such languages

References

[1] Guillaume Allais et al. "A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236785. URL: https://doi.org/10.1145/3236785.

[2] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0.

[3] G.P. Huet. "A unification algorithm for typed λ-calculus". In: *Theoretical Computer Science* 1.1 (1975), pp. 27–57. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(75)90011-0.

[4] Daniel R. Licata and Robert Harper. "2-Dimensional Directed Type Theory". In: *Electronic Notes in Theoretical Computer Science* 276 (2011). Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII), pp. 263–289. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2011.09.026. URL: https://www.sciencedirect.com/science/article/pii/S1571066111001174.

[5] Andres Löh, Conor McBride, and Wouter Swierstra. "A Tutorial Implementation of a Dependently Typed Lambda Calculus". In: *Fundamenta Informaticae* 102 (Jan. 2010), pp. 177–207. DOI: 10.3233/FI-2010-304.

[6] Dale Miller. "A logic programming language with lambda-abstraction, function variables, and simple unification". In: *Extensions of Logic Programming*. Ed. by Peter Schroeder-Heister. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 253–281. ISBN: 978-3-540-46879-0.

[7] Emily Riehl and Michael Shulman. "A type theory for synthetic ∞-categories". In: *Higher Structures* 1.1 (2017), pp. 116–193. arXiv: 1705.07442.

[8] Wayne Snyder. "Higher order E-unification". In: *10th International Conference on Automated Deduction*. Ed. by Mark E. Stickel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 573–587. ISBN: 978-3-540-47171-4.

[9] Wouter Swierstra. "Data types à la carte". In: *Journal of Functional Programming* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

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

[11] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341691. URL: https://doi.org/10.1145/3341691.

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

[13] Janis Voigtländer. "Asymptotic Improvement of Computations over Free Monads". In: July 2008, pp. 388–403. ISBN: 978-3-540-70593-2. DOI: 10.1007/978-3-540-70594-9_20.

[14] Nicolas Wu and Tom Schrijvers. "Fusion for Free". In: *Mathematics of Program Construction*. Ed. by Ralf Hinze and Janis Voigtländer. Cham: Springer International Publishing, 2015, pp. 302–322. ISBN: 978-3-319-19797-5.

Zaionc paradox revisited (Abstract)

Pierre Lescanne

University of Lyon, École Normale Supérieure de Lyon,
LIP (UMR 5668 CNRS ENS Lyon UCBL),
46 allée d'Italie, 69364 Lyon, France
pierre.lescanne@ens-lyon.fr

In 2007, Marek Zaionc coauthored two papers [3, 2], corresponding to two models of the calculus of implicative propositions and presenting the following paradox, namely that *asymptotically almost all classical theorems are intuitionistic*, which is called here *Zaionc paradox*. In the current paper, we focus on the model of [3], which we call *canonical expressions*. They have been introduced by Genitrini, Kozik and Zaionc [3] and more recently by Tarau and de Paiva [9, 10]. A canonical expression is a representative of a class of implicative expressions that differ only by the name assigned to the variables. Whereas Genitrini, Kozik and Zaionc addressed the mathematical aspect of this model, Tarau and de Paiva tried to explicitly generate all the canonical expressions of a given size and faced up to combinatorial explosion, because canonical expressions grow super exponentially in size. In this paper, I check experimentally Zaionc paradox, adopting a Monte-Carlo approach to observe how this paradox emerges. Indeed I designed a linear algorithm to randomly generate canonical expressions. Therefore I can consider large samples of random canonical expressions and count how many canonical expressions in that samples are intuitionistic theorems or classical theorems. A long version of this document is [6] and the programs used in this paper can be found on GitHub.

The model of canonical expressions

We call *canonical expression* the representative of an equivalence class of binary expressions up-to renaming of variables. In other words, a canonical expression is a binary expression, in which variables are named canonically, from right to left. That means that the rightmost variable is x_0, then if processing to the left, the next new variable is x_1, then the next new variable, which is neither x_0 nor x_1 is x_2 etc.

Since canonical expressions are pairs of well-known combinatorial objects, namely binary trees and congruence classes, we can use well-known algorithm to generate each constituents of the pairs.

Random binary trees For generating random binary trees I use *Rémy algorithm* [7] which is linear. This algorithm is described by Knuth in [4] § 7.2.1.6 (pp. 18-19). I have taken his implementation. The idea of the algorithm is that a random binary tree can be built by iteratively and randomly picking a node or a leaf in a random binary tree and inserting a new leaf either on the left or on the right.

Random restricted growth string For generating random partitions or random restricted growth strings an algorithm due to A. J. Stam [8] and described by Knuth in [5] § 7.2.1.3 (p. 74) was implemented.

Selecting intuitionistic theorems

Once a canonical expression is randomly generated, one has to check whether it is an intuitionistic theorem, a classical theorem, or not a theorem of those sorts. For that, heuristics are applied.

Simple intuitionistic theorems A *simple intuitionistic theorem* is a theorem, in which the goal is among the premises.

Elim intuitionistic theorems Let us call *Elim intuitionistic theorem* (for $\rightarrow$-*eliminating* theorem), a theorem which is a direct application of the modus ponens aka $\rightarrow$-elimination.

Easy intuitionistic theorems Let us call *Easy intuitionistic theorems*, expressions that are simple or Elim.

Removing trivial premises In intuitionistic logic if a premise is a theorem, it can be removed.

Silly intuitionistic theorems A silly theorem is a theorem of the form $... \rightarrow p \rightarrow ... \rightarrow p$. Detecting such expressions has a cost, I decided to no detect silly intuitionistic theorem recursively by only after easy subexpressions have been removed recursively.

Cheap intuitionistic theorems Let us call *cheap intuitionistic theorems*, expressions that are silly or easy after removing (recursively) easy premises.

Classical tautologies

The selection of classical tautologies is as usual, by valuations. Indeed if all the valuations of a given expression yield `True` this expression is a classical tautology. But this method is obviously intractable [1]. It should be applied only to expressions on which other more efficient methods do not work and with a limitation of the number of variables in expressions[1].

Trivial non classical propositions Before applying valuations, some trivial non classical propositions must be eliminated. An expression e is trivially non classical if its premises have a goal which is not x_0 or are simple with goal x_0.

Results

I run my Haskell program on a sample of 20 000 randomly generated canonical expressions of size 100 and I found 759 classical tautologies, among which 733 were cheap expressions, hence guaranteed to be intuitionistic theorems. Therefore the ratio of cheap theorems over classical theorems is 96.6%. Are the 26 classical non cheap theorems still intuitionistic? The experience cannot tell. I presume that there are likely more than 733 intuitionistic theorems and therefore more than 96.6% of classical theorems that are intuitionistic.

Conclusion

Algorithms for random generation presented in *The Art of Computer Programming* [4, 5] allow implementing Monte-Carlo methods that confirm experimentally Zaionc paradox and show that the convergence[2] of the set of intuitionistic theorems toward this of classical theorems is faster than expected from the asymptotic approximations proposed by the theory [3]. Indeed, whereas I compare the set of cheap intuitionistic theorems with this of classical theorems, Genitrini, Kozik and Zaionc compare the set of simple intuitionistic theorems with the set of non simple non tautologies. This is a too rough approximation. This suggests to complete the analytic development to justify this faster convergence.

[1] In my experience with canonical expressions of size 100 the variables should not exceed 30 which is rare enough to affect no classical theorem.

[2] As the size of the expressions grows.

References

[1] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. URL: https://doi.org/10.1145/800157.805047, doi:10.1145/800157.805047.

[2] Hervé Fournier, Danièle Gardy, Antoine Genitrini, and Marek Zaionc. Classical and intuitionistic logic are asymptotically identical. In Jacques Duparc and Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 177–193. Springer, 2007.

[3] Antoine Genitrini, Jakub Kozik, and Marek Zaionc. Intuitionistic vs. classical tautologies, quantitative comparison. In Marino Miculan, Ivan Scagnetto, and Furio Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, volume 4941 of *Lecture Notes in Computer Science*, pages 100–109. Springer, 2007. URL: https://doi.org/10.1007/978-3-540-68103-8_7, doi:10.1007/978-3-540-68103-8_7.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Publishing Company, 2005.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees, History of Combinatorial Generation*. Addison-Wesley Publishing Company, 2006.

[6] Pierre Lescanne. Zaionc paradox revisited. HAL Archives ouvertes, 2021. URL: https://hal.archives-ouvertes.fr/hal-03197423v1.

[7] Jean-Luc Rémy. Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire. *RAIRO Theor. Informatics Appl.*, 19(2):179–195, 1985. URL: https://doi.org/10.1051/ita/1985190201791, doi:10.1051/ita/1985190201791.

[8] A. J. Stam. Generation of a random partition of a finite set by an urn model. *J. Comb. Theory, Ser. A*, 35(2):231–240, 1983. URL: https://doi.org/10.1016/0097-3165(83)90009-2, doi:10.1016/0097-3165(83)90009-2.

[9] Paul Tarau. A hiking trip through the orders of magnitude: Deriving efficient generators for closed simply-typed lambda terms and normal forms. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2016. URL: https://doi.org/10.1007/978-3-319-63139-4_14, doi:10.1007/978-3-319-63139-4_14.

[10] Paul Tarau and Valeria de Paiva. Deriving theorems in implicational linear logic, declaratively. In Francesco Ricca, Alessandra Russo, Sergio Greco, Nicola Leone, Alexander Artikis, Gerhard Friedrich, Paul Fodor, Angelika Kimmig, Francesca A. Lisi, Marco Maratea, Alessandra Mileo, and Fabrizio Riguzzi, editors, *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020*, volume 325 of *EPTCS*, pages 110–123, 2020. URL: https://doi.org/10.4204/EPTCS.325.18, doi:10.4204/EPTCS.325.18.

A Practical Implementation of Twin Types

Víctor López Juan

Chalmers University of Technology, Gothenburg, Sweden
victor@lopezjuan.com

Abstract

In a previous publication [8], an approach to higher-order unification in a dependently-typed setting is described and benchmarked on a specific case study by means of a prototype. In this follow-up we evaluate the practicality of our approach by implementing it in an existing proof assistant (Agda), and benchmarking it on a selection of projects, including the Agda standard library. This solves some bugs in the current Agda implementation, with no large effect on performance and a limited amount of changes to the code of Agda.

When defining functions in a dependently-typed proof assistant, users may mark some function arguments as implicit, either for readability or for code writing speed. The type checker needs to infer these arguments when the function is used.

Inference of implicit arguments in proof assistants such as Coq, Lean, Idris or Agda is done by replacing the implicit arguments with placeholders of an appropriate type (i.e. metavariables), and applying the typing rules to produce a series of unification constraints. A unification constraint consists of at least a pair of terms t and u in a typing context Γ ($\Gamma \vdash t \approx u$). A constraint is solved by assigning terms to the placeholders so that both sides of the constraint become equal ($\Gamma \vdash t \equiv u$). A constraint of the form $\Gamma \vdash \alpha \approx t$ can be solved by assigning $\alpha := t$. The creation of ill-typed terms can be avoided by either check that t has the right type before assigning it to α, which could incur a performance penalty; or by ensuring that both sides of every constraint have the same context and type, which is the preferred approach.

Binder problem A common problem in dependently-typed unification arises when unifying binders. For instance, a constraint of the form $\Gamma \vdash (x : A) \to B \approx (x : A') \to B'$ may be solved by separately unifying the domains and the codomains. However, the types B and B' live in different contexts ($\Gamma, x : A \vdash B$ **type** and $\Gamma, x : A' \vdash B'$ **type**). It is not self-evident what the type of x should be in $\Gamma, x : ? \vdash B \approx B'$ **type**.

Spine problem Consider an irreducible constant $c : (x : A) \to (y : B) \to C$ (e.g. a data constructor). The spine problem arises for instance when solving $\Gamma \vdash c\, t\, u \approx c\, t'\, u'$, which is done by unifying $\Gamma \vdash t \approx t' : A$ and $\Gamma \vdash u \approx u'$. Until t and t' can be unified, u and u' may lack a common type. This may threaten the soundness of ensuing metavariable instantiations.

Coq In our tests, Coq [17] will refuse to unify the domain before the codomain, thus avoiding the binder problem. Ziliani and Sozeau [18] argue that in the context of Coq, constraint postponement is not crucial, and may even worsen the performance of the algorithm and make it harder to debug. The Lean proof assistant behaves similarly to Coq. The spine problem in Coq is also avoided by solving the constraints in a suitable order.

Agda In Agda constraints are well-typed modulo other constraints being solved. The presence of ill-typed terms may however cause issues, as described by Norell and Coquand [12, 14]. To mitigate them, the type of the variable is replaced by a blocked constant. That is, $\Gamma, x : p \vdash B \approx B'$, where p reduces to A ($p \rightsquigarrow A$) when $\Gamma \vdash A \equiv A'$. The resulting terms are still

		CPU (s)		Memory (MB)	
	kLOC	Ours	Baseline	Ours	Baseline
Prelude	12.6	110±1.2 (+5.1…+1.8%)	106±1.2	573±15 (+6.0…−1.8%)	562±17
Std-Lib	93.6	531±7.8 (+0.7…−2.8%)	537±5.3	1690±12 (−3.7…−5.3%)	1770± 8
HoTT	29.7	223±2.2 (+5.9…+3.1%)	214±2.1	1910±36 (−2.6…−7.4%)	2010±35

Table 1: Performance when type-checking some Agda projects, namely the Agda prelude [15], the Agda standard library [4], and "Introduction to HoTT" [16]. The size in thousands of lines of code is given, followed by estimations of the median time and memory used by our implementation [9] and the percentual variation with respect to the version of Agda it is based on [2] ($n = 40$, 95% CI). We observe no large differences between the two implementations.

potentially not well-typed, which in some known cases causes the type-checker to crash [10]. Agda mitigates the spine problem in a similar way, but it is not a complete solution [3, 13].

Idris 2 Idris 2 is a programming language focused on type-driven development. The binder problem is solved by taking $x : A$, and replacing all occurrences of x in B' by a blocked constant; i.e. $\Gamma, x : A \vdash B \approx B'[p/x]$ **type**, where $p \rightsquigarrow x$ when $\Gamma \vdash A \equiv A'$. As opposed to replacing the type of the variable, as done in Agda, blocking the variable produces well-typed terms. This straightforward approach was however not flexible enough to support some existing Agda code [1], and does not address the spine problem.

An approach based on twin types Gundry and McBride [6, 5] propose a solution based on assigning two types to each variable, and annotating each occurrence of the variable in a term to distinguish which type applies. A streamlined variant of their approach without the annotations [8, 7] has been implemented in an existing prototype [11] and used to type-check some specific examples. It remained to show that the approach scales to a larger proof assistant.

Evaluation results We have implemented the method we described in previous publications [8, 7] (with suitable extensions) into the Agda type checker [2]. The implementation took 18 weeks of work at 25 hours per week. Less than 2700 lines of code (6.7% of the codebase) were added or modified, partly thanks to keeping the term syntax intact. Implementing our approach in Agda fixes some long-standing bugs either outright [10] or without resorting to workarounds [3, 13]. We have tested our implementation [9] on three large Agda projects, yielding comparable CPU and memory usage (Table 1). Note that hash consing, which was used for performance in a previous prototype [8, 7], was not required here.

Limitations This approach does not preserve all the power of the existing Agda implementation. For instance, four instances of implicit arguments in the standard library which were previously inferred by Agda had to be given explicitly. For other projects with different coding styles, the figure may be larger. Support for Agda's cubical type theory features is pending.

Conclusion Despite the limitations, heterogeneous higher-order unification is a viable approach for type-directed inference of unique solutions for implicit arguments. Performance is comparable to the approach used in Agda, the amount of changes to the Agda implementation is limited, and some long-standing bugs in Agda are fixed.

References

[1] Andreas Abel, Jesper Cockx, Nils Anders Danielsson, and Víctor López Juan. Regression related to fix of #3027. Agda Issue #4408, January 2020.

[2] Andreas Abel, Nils Anders Danielsson, Ulf Norell, et al. Agda, 2021. Commit 47179128.

[3] Andreas Abel, Martin Stone Davis, Ulf Norell, et al. (No longer an) Internal error at src/full/Agda/TypeChecking/Substitute.hs:98. Agda Issue #2709, August 2017.

[4] Nils Anders Danielsson, Matthew Daggitt, Guillaume Allais, et al. The Agda standard library, 2021. Commit `d00ba755c`, file "EverythingSafe.agda".

[5] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, Department of Computer and Information Sciences, University of Strathclyde, 2013.

[6] Adam Gundry and Conor McBride. A tutorial implementation of dynamic pattern unification. Unpublished, 2012.

[7] Víctor López Juan. *Practical Unification for Dependent Type Checking*. Licentiate Thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2020.

[8] Víctor López Juan and Nils Anders Danielsson. Practical dependent type checking using twin types. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2020, 2020.

[9] Víctor López Juan et al. Agda – pull request #5234, 2021. Commit bc37b6156.

[10] Víctor López Juan and Ulf Norell. Internal error in the presence of unsatisfiable constraints. Agda Issue #3027, April 2018.

[11] Francesco Mazzoli, Nils Anders Danielsson, Ulf Norell, Andrea Vezzosi, Andreas Abel, et al. Tog – a prototypical implementation of dependent types, 2017.

[12] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2007.

[13] Ulf Norell. Internal error during instance search. Agda Issue #3870, June 2019.

[14] Ulf Norell and Catarina Coquand. Type checking in the presence of meta-variables. Unpublished, 2007.

[15] Ulf Norell et al. The Agda prelude, 2020. Commit `fcca646d6`, file "Everything.agda".

[16] Egbert Rijke. *Introduction to Homotopy Type Theory*. 2020.

[17] The Coq Development Team. Coq 8.13.1, March 2021.

[18] Beta Ziliani and Matthieu Sozeau. A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. *Journal of Functional Programming*, 27, 2017.

Functorial Adapters

Conor Mc Bride and Fredrik Nordvall Forsberg

University of Strathclyde, Glasgow, Scotland

We present some progress towards a calculus in which container-like data structures exhibit functoriality by construction, equipped with an equational theory in which functor laws hold on the nose. Specifically, we work in a *bidirectional* setting [PT00, DK20, McB], where types for introduction forms are checked ($T \ni t$) and elimination forms have types synthesized ($e \in S$).[1] To embed the latter in the former, we have formerly given the 'change of direction' rule

$$\frac{e \in S \quad S = T}{T \ni e}$$

but we have now come to see this as a missed opportunity. This is the one place where we know both the type we have and the type we want, so we might exploit this richness of information to explore more interesting ways to get from one to the other than making sure the types match and doing nothing: that is but the *identity* for a category of *adapters* which offer functorial actions on data.

Our intention is to build a dependent type theory this way, but the concept already makes sense in a simply typed setting, so let us start there, for pedagogical purposes. Types are given by the following grammar:

$$\sigma, \tau \quad ::= \quad \mathbf{1} \quad | \quad \sigma \times \tau \quad | \quad \sigma \to \tau \quad | \quad \tau^*$$

Our terms are split between the *constructions* and the *computations*, but at the point where they meet, we now allow an *adapter* which can be blank.

$$\begin{array}{rrcl}
\text{construction} & s, t & ::= & \star \mid (s,t) \mid \lambda x.\, t \mid [] \mid [t] \mid s +\!\!+ t \mid a\, e \\
\text{computation} & e, f & ::= & x \mid \mathsf{outl}\, e \mid \mathsf{outr}\, e \mid f\, t \mid \mathsf{foldr}_\tau\, t\, (x.\, y.\, s)\, e \mid (t : \tau) \\
\text{adapter} & a, b & ::= & \mid s \times t \mid s \to t \mid t^*
\end{array}$$

Note that we *construct* lists with $+\!\!+$ with intent to enforce the equational theory of free monoids.

As well as the blank identity adapter, we respect functoriality of type constructors by offering mapping over them as an adapter. The 'change of direction' now becomes

$$\frac{e \in \sigma \quad \sigma \,|a\rangle\, \tau}{\tau \ni a\, e} \qquad \frac{}{\sigma \,|\rangle\, \sigma} \qquad \frac{\sigma \to \sigma' \ni s \quad \tau \to \tau' \ni t}{\sigma \times \tau \,|s \times t\rangle\, \sigma' \times \tau'} \qquad \frac{\sigma' \to \sigma \ni s \quad \tau \to \tau' \ni t}{\sigma \to \tau \,|s \to t\rangle\, \sigma' \to \tau'} \qquad \frac{\sigma \to \tau \ni t}{\sigma^* \,|t^*\rangle\, \tau^*}$$

and the fact that we know both the source and the target type for the adapter gives us an easy way to check the action on elements without resorting to type annotation or guesswork. For reasons of space, we will focus on list adapters in the rest of this abstract.

The spirit of bidirectional type systems is that types are *present* when there is work to be done, but vanish from normal forms. We call a term $(t : \tau)$ a *radical* by analogy with organic chemistry, because its type gives it the power of computation. (We present an equational theory which is also bidirectionally typed.) E.g., in the β-rule for functions, a function type is broken

[1] Diverse notations for bidirectional typing judgements abound in the literature: we choose to keep time flowing left to right. We omit global contexts but give local context extensions.

down and its pieces used to create radicals which may compute further. Radicals lose their types when computation is finished.

$$\frac{}{(\lambda x.\, t : \sigma \to \tau)\, s = (t\{(s:\sigma)/x\} : \tau) \in \tau} \qquad \frac{}{\sigma \ni (s:\sigma) = s}$$

When adapters collide, there is necessarily a 'type in the middle', which normalisation must remove. Hence it is vital that adapters compose. Fortunately, the point of functoriality is to respect composition. Identity adapters are absorbed and maps fuse.

$$\frac{e \in \rho \quad \rho\,|a\rangle\,\sigma\,|b\rangle\,\tau = \rho\,|c\rangle\,\tau}{\tau \ni b\,(a\,e : \sigma) = c\,e} \qquad \frac{}{\rho^*\,|s^*\rangle\,\sigma^*\,|t^*\rangle\,\tau^* = \rho^*\,|(\lambda x.\,(t:\sigma\to\tau)\,((s:\rho\to\sigma)\,x))^*\rangle\,\tau^*}$$

The type in the middle, σ^*, is removed by composition, but its components move to potential β-reducts within the fused adapter. The way adapters must compose is strongly reminiscent of the way transitivity must be admissible in coercive subtyping [Luo99], and indeed, the blank adapter might very well generalise to coercions rather than just identity, especially with functor laws now holding as envisaged by Adams and Luo [LA08].

Of course, adapters act on constructions, making use of both types.

$$\frac{}{\tau^* \ni t^*\,([] : \sigma^*) = []} \qquad \frac{}{\tau^* \ni t^*\,([s] : \sigma^*) = [(t:\sigma\to\tau)\,s]}$$

$$\frac{}{\tau^* \ni t^*\,(ss +\!\!+ ss' : \sigma^*) = t^*\,(ss : \sigma^*) +\!\!+ t^*\,(ss' : \sigma^*)}$$

However, they may also fuse with eliminations, acting elementwise in the step case.

$$\frac{e \in \rho^*}{\mathsf{foldr}_\tau\, t_n\,(x.\,y.\,t_c)\,(t^*\,e : \sigma^*) = \mathsf{foldr}_\tau\, t_n\,(w.\,y.\,t_c\{(t:\rho\to\sigma)\,w/x\})\,e \in \tau}$$

Normalisation thus pulls mapping towards the change of direction, inwards through constructions by their action, and outwards through computations by fusion and potentially naturality.

We have implemented a decision procedure for our equational theory as a two phase process in the style of normalization-by-evaluation, with type information flowing the same way as in the typing rules: β-reduction and action on constructors happen in the first phase, η-expansion in the second. We have some flexibility about when to compose adapters and test for identity:

$$\frac{x : \tau \vdash \tau \ni (t : \tau \to \tau)\,x = x}{\tau^*\,|t^*\rangle\,\tau^* \;=\; \tau^*\,|\rangle\,\tau^*}$$

We chose to do the former in the first phase and the latter (with its attendant type-directed η-expansion) in the second. E.g., if $\tau = \mathbf{1}$, t is certainly the identity pointwise by the η-rule for $\mathbf{1}$, so any mapping adapter must be the identity. Meanwhile, mapping swapping for nontrivial $\sigma \times \tau$ is not the identity, but when two map-swap adapters compose, they yield the identity.

The upshot is that the usual $\beta\eta$-equality is extended with monoid, functor and map-fusion laws for lists, rationalising earlier work on tidying neutral terms [AMB13], adding clarity about the flow of type information. We thus step beyond the old habit of implementing algorithmic equality for open terms by just letting an untyped evaluator for closed terms get stuck at free variables. While type erasure remains desirable for the execution of closed programs, the presence of type information in open computation is a currently underexploited resource from which we are only beginning to profit.

Knocking at the door is the fact that there are categories other than that of types and functions, and thus that datatypes could always yield functorial adapters, with respect to at least equality structure. Our current notions of dependent inductive datatype respect at most equality. Adapters show that we can respect nontrivial categorical structure intensionally and by construction, when we do so deliberately.

References

[AMB13] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In Stephanie Weirich, editor, *Dependently-typed programming '13*, pages 13–24. ACM, 2013.

[DK20] Jana Dunfield and Neel Krishnaswami. Bidirectional typing, 2020. arxiv:1908.05839.

[LA08] Zhaohui Luo and Robin Adams. Structural subtyping for inductive types with functorial equality rules. *Mathematical Structures in Computer Science*, 18(5):931–972, 2008.

[Luo99] Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[McB] Conor McBride. The types who say 'ni'. Forthcoming, draft available at https://github.com/pigworker/TypesWhoSayNi.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Language Systems*, 22(1):1–44, 2000.

Towards Automation for Iris

Ike Mulder[1] and Robbert Krebbers[1]

Radboud University of Nijmegen

Recent tools for separation logic have enabled formal verification of challenging fine-grained concurrent programs. However, currently one needs to choose between tools with good automation [3, 8, 9], and tools that are foundational and expressive [5, 7].

An example of a tool with good automation is Caper [3]. While Caper can be used to verify challenging fine-grained concurrent programs with little user assistance, its trusted code base is large. Caper is a standalone tool (written in Haskell) whose logic has not been mechanized, and which uses SMT solvers as trusted oracles. On the other hand, there is Iris [5]—a framework for (higher-order) concurrent separation logic embedded in the Coq proof assistant. While Iris is expressive, proofs are mostly manual: they are developed interactively using tactics that come with the Iris Proof Mode [6]. The trusted code base for Iris is small: only Coq's proof checker and the programming language's operational semantics need to be trusted.

We aim to arm Iris with strong automation to obtain the best of both worlds. Concretely, we are developing tactics that entirely solve easy goals, and make good partial progress on difficult goals. This means we are looking for automation that is *goal directed* (to ensure efficiency), and that does *not rely on backtracking* (so we do not end up in an unprovable state).

Iris is a challenging target for automation because of its expressive logic. Particularly, Iris comes with *ghost resources* to model protocols on (concurrent) data structures. Ghost resources are similar to the points-to resource $\ell \mapsto v$, but they are not tied to the syntax of the program. A ghost resource with name γ and value a is denoted as $\boxed{a}^{\gamma}$. Changes to ghost resources are handled through the 'fancy update' ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$, a modality which behaves like a strong indexed monad. Furthermore, Iris comes with *invariants* for sharing resources. Invariants with name $\mathcal{N}$ and resource P are denoted as $\boxed{P}^{\mathcal{N}}$. Invariants $\boxed{P}^{\mathcal{N}}$ can be duplicated (*i.e.*, shared between threads), but ownership of P can only be obtained for the duration of atomic steps.

The expressivity of Iris plays against us, as many of its proof rules are not goal directed. In various Iris proofs, ghost resources are manipulated to precisely the right state, just before it becomes apparent that this coincides with the proof obligation. At that point, the manipulation is often no longer possible. A selection of Iris's proof rules is shown in Figure 1.

$$\textsc{ghost-update}\quad \frac{a \rightsquigarrow b \qquad \boxed{b}^{\gamma} \vdash \Rightarrow Q}{\boxed{a}^{\gamma} \vdash \Rightarrow Q}$$

$$\textsc{wp-faa}\quad \frac{\ell \mapsto (n+m) \vdash Q}{\ell \mapsto n \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{FAA}(\ell, m)\, \{Q\}}$$

$$\textsc{wp-atomic}\quad \frac{\mathsf{atomic}(e)}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e \left\{ {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_1} Q \right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{Q\}}$$

$$\textsc{inv-open1}\quad \frac{\mathsf{atomic}(e) \qquad \boxed{P}^{\mathcal{N}} * \triangleright P \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\triangleright P * Q\}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{wp}_{\mathcal{E} \uplus \mathcal{N}}\, e\, \{Q\}}$$

$$\textsc{inv-open2}\quad \frac{\triangleright P * (\triangleright P \;{}^{\mathcal{E}_1}\!\Rightarrow\!\!*^{\mathcal{E}_1 \uplus \mathcal{N}} \mathsf{True}) \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}_1 \uplus \mathcal{N}}\!\Rightarrow^{\mathcal{E}_2} Q}$$

Figure 1: A selection of Iris's proof rules.

Example and approach. Suppose we wish to verify the increment operation of a counter module. We will be working in a language with at least an atomic Fetch And Add (FAA) instruction operating on locations in a heap. If the value of the counter is stored in location ℓ, we can define $\mathsf{incr}(\ell) := \mathsf{FAA}(\ell, 1)$. The implementation of our module is thus very simple: it uses the available FAA instruction to atomically increment the value at location ℓ by 1. We can prove the specification $\ell \mapsto n \vdash \mathsf{wp}\ \mathsf{incr}(\ell)\ \{\ell \mapsto (n+1)\}$, but this is not very useful. It will not allow us to prove that executing two increments in parallel increments the counter by 2, since that specification requires both threads to have exclusive access to $\ell \mapsto n$.

A more useful specification would be as follows, where $q \in (0, 1] \cap \mathbb{Q}$:

$$I := \exists n.\, \ell \mapsto n * \boxed{\bullet n}^{\gamma}, \qquad \boxed{I}^{\mathcal{N}} * \boxed{\circ(q, m)}^{\gamma} \vdash \mathsf{wp}\ \mathsf{incr}(\ell)\ \left\{\boxed{\circ(q, m+1)}^{\gamma}\right\}.$$

Here $\boxed{\bullet \cdot}$ and $\boxed{\circ(\cdot, \cdot)}$ are some of Iris's ghost resources, and in this context $\boxed{\circ(q, m)}^{\gamma}$ will mean "some thread has witnessed the counter to be incremented with m". The specification above, together with this property of $\boxed{\circ(q, m)}^{\gamma}$, makes it possible to prove that executing two increments in parallel increases the counter by 2.

So how would a typical proof of this specification in Iris go? The FAA instruction requires an $\ell \mapsto -$ resource, which is currently not available. Therefore, we first use INV-OPEN1 to take the resource I out of the invariant $\boxed{I}^{\mathcal{N}}$. After eliminating the existential, we are in shape to apply WP-FAA. But if we look ahead a bit, we see that our physical resource $\ell \mapsto (n+1)$ will be out of sync with our logical resource $\bullet n$, meaning that we will not be able to restore I. Therefore, we first use GHOST-UPDATE to update $\boxed{\bullet n}^{\gamma} * \boxed{\circ(q, m)}^{\gamma}$ to $\boxed{\bullet(n+1)}^{\gamma} * \boxed{\circ(q, (m+1))}^{\gamma}$. After this step we can apply WP-FAA, restore invariant I and prove the postcondition.

We have devised a proof-search algorithm, implemented as a Coq tactic, that can automatically prove the above specification. We use ideas from logic programming [2, 4] and bi-abduction [1]. Our key ideas are:

- We rewrite Iris's proof rules to make them amenable for goal-directed search
- We postpone introduction of existentials and logical updates as long as possible
- We use bi-abduction (keying both on the goal and hypotheses) to find the correct proof rules

In the example above, our algorithm uses a derived rule combining WP-ATOMIC and WP-FAA to go to an intermediate goal:

$$\boxed{I}^{\mathcal{N}} * \boxed{\circ(q, m)}^{\gamma} \vdash {}^{\top}\!\Rrightarrow^{?} \exists n.\, l \mapsto n * (l \mapsto (n+1) \ {}^{?}\!\Longrightarrow^{\top} \boxed{\circ(q, (m+1))}^{\gamma}).$$

It is crucial that we do not introduce the $\exists\, n$ in the goal; at this point we do not know the n for which $\ell \mapsto n$, since it is still existentially quantified in the invariant I. The bi-abduction part of our proof search then finds that $\boxed{I}^{\mathcal{N}} \vdash {}^{\top}\!\Rrightarrow^{\top \setminus \mathcal{N}} \exists n.\, l \mapsto n * ?$, by applying INV-OPEN2 and eliminating the existential. The proof can then continue by using the acquired extra ? resource, and the same bi-abduction algorithm can later infer the required update from $\bullet n$ to $\bullet(n+1)$, which will require the $\circ(q, m)$ resource, produce a $\circ(q, m+1)$ resource, and finish the proof.

The tactic additionally performs no global backtracking. We use backtracking inside the algorithm to find lemmas, but stick with any lemma we find. This works because we write lemmas in a particular format, such that there is exactly one way to make progress.

Conclusion. Our tactic is able to prove correct various small but tricky programs that use fine-grained concurrency. For example, the specifications of a spin lock, ticket lock, counter module and concurrent stack can all be proven with little user assistance. The implementation relies on Coq's type class mechanism to find appropriate lemma's, and is built on top of the Iris Proof Mode. Our next steps will be a thorough comparison with other tools, larger case studies, and an investigation into dealing with notions like logical atomicity.

References

[1] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[2] I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *TCS*, 232(1-2):133–163, 2000.

[3] T. Dinsdale-Young, P. da Rocha Pinto, K. J. Andersen, and L. Birkedal. Caper - automatic verification for fine-grained concurrency. In *ESOP*, volume 10201 of *LNCS*, pages 420–447. 2017.

[4] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *I&C*, 110(2):327–365, 1994.

[5] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP*, 28:e20, 2018.

[6] R. Krebbers, J.-H. Jourdan, R. Jung, J. Tassarotti, J.-O. Kaiser, A. Timany, A. Charguéraud, and D. Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, 2(ICFP):1–30, 2018.

[7] W. Mansky, A. W. Appel, and A. Nogin. A verified messaging system. *PACMPL*, 1(OOPSLA):87:1–87:28, 2017.

[8] A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. In *TACAS*, volume 10805 of *LNCS*, pages 190–209, 2018.

[9] F. A. Wolf, M. Schwerhoff, and P. Müller. Concise outlines for a complex logic: A proof outline checker for tada, 2020. Manuscript, available at https://arxiv.org/abs/2010.07080.

Quantitative polynomial functors*

Georgi Nakov and Fredrik Nordvall Forsberg

University of Strathclyde, Glasgow, U.K.
{georgi.nakov,fredrik.nordvall-forsberg}@strath.ac.uk

Abstract

We investigate containers and polynomial functors in Quantitative Type Theory and give initial algebra semantics of inductive data types in presence of linearity. Reasoning by induction is supported and equivalent to initiality also in the linear setting.

Quantitative Type Theory (QTT) Quantitative Type Theory [Atk18, McB16] combines linear and dependent types, allowing for tracking and reasoning about resource usage of programs. Such a combination is non-trivial as there is no obvious answer how to treat terms occurring in type formation. Previous attempts include the Linear Logical Framework [CP02], and [KPB15], based on Benton's Linear/Non-Linear logic [Ben95], in which the context is split into intuitionistic and linear parts and each type is allowed to depend on only one of the two. QTT differs by maintaining a single context where each variable is annotated with resource information. For example, consider the judgement:

$$n \overset{0}{:} \mathsf{Nat}, x \overset{2}{:} \mathsf{Fin}(n), f \overset{1}{:} \mathsf{Fin}(n) \overset{2}{\to} \mathsf{Fin}(n) \vdash f(x) : \mathsf{Fin}(n)$$

The quantities on the left of the turnstile (taken from a fixed semiring R) denote how many times the variables must be used in the scope. The term on the right is tacitly annotated with 1 to denote computational relevance. The core insight is that contemplating variables in a type is always possible, even for already consumed ones. Thus we get unrestricted usage in type formation and semiring-controlled usage for terms present at run time. Evident in the previous example is also that resource tracking is integral to type formers in the system, e.g. a function f of type $f : (x \overset{\rho}{:} A) \to B$ must be supplied with ρ many copies of its argument, and the second component of the dependent pair $t : (x \overset{\rho}{:} A) \otimes B$ requires ρ many copies of the first component.

Polynomial functors An ad-hoc approach of writing out introduction and elimination rules for each data type is cumbersome and error-prone. A principled solution to adding inductive data types in a traditional setting is provided by the theory of polynomial functors [HJ98] and containers [AAG03]. We build the syntactic category of closed types and linear functions in QTT and define polynomial functors on it. We can then systematically derive the appropriate elimination rule for an initial algebra of the functor.

Proposition 1. *Let $\mathcal{C}$ be the category of closed types and linear functions: its objects are types $\vdash X$, and morphisms are functions $\vdash f : X \overset{1}{\to} Y$. For fixed $\vdash A$ type and $x \overset{0}{:} A \vdash B$ type, the mapping $F_{A,B}(X) = (a \overset{1}{:} A) \otimes (B[a] \overset{1}{\to} X)$ is a functor $\mathcal{C} \to \mathcal{C}$.*

We call any functor isomorphic to one of the form $F_{A,B}$ a *quantitative container*. The above proposition can be generalized to types and functions over an arbitrary, fixed context of the shape $\Gamma = 0\Gamma$, i.e. where all variables are annotated with 0.

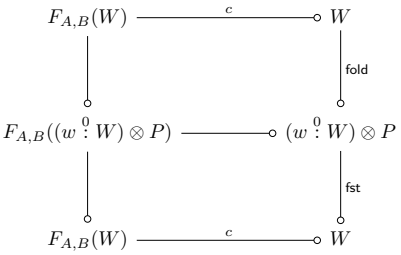
*Supported by the UK National Physical Laboratory Measurement Fellowship project *Dependent types for trustworthy tools*.

Induction principles from initiality Recall that an F-algebra for an endofunctor $F : \mathcal{C} \to \mathcal{C}$ is a pair $(A, a : F(A) \to A)$, where A is an object of $\mathcal{C}$ and a is a $\mathcal{C}$-morphism. A morphism between F-algebras (A, a) and (B, b) is a map $f : A \to B$ in $\mathcal{C}$, such that $f \circ a = b \circ F(f)$. We now turn to the equivalence of the induction principle and existence of an initial algebra:

Theorem 2. *Let* $\mathbf{W} := (W, c : F_{A,B}(W) \xrightarrow{1} W)$ *be an* $F_{A,B}$*-algebra.* $\mathbf{W}$ *is initial iff the following induction principle holds:*

$$\frac{w \overset{0}{:} W \vdash P \quad \vdash M : (a \overset{1}{:} A) \to (h \overset{0}{:} B[a] \xrightarrow{1} W) \to ((b \overset{1}{:} B[a]) \to P(h(b))) \xrightarrow{1} P(c(a,h))}{\vdash \mathsf{elim}(P, M) : (x \overset{1}{:} W) \to P[x]}$$

Proof (sketch). We alter the standard construction [AGS17], emphasizing on the role of linearity. Assuming that $\mathbf{W}$ is initial, build an $F_{A,B}$-algebra for the dependent tensor type $(w \overset{0}{:} W) \otimes P$ and get the unique mediating morphism $\mathsf{fold} : W \xrightarrow{1} (w \overset{0}{:} W) \otimes P$ by initiality. Compose with the second projection $\mathsf{snd} : (x \overset{1}{:} (w \overset{0}{:} W) \otimes P) \to P[\mathsf{fst}(x)]$ to get a map $(x \overset{1}{:} W) \to P[\mathsf{fst}(\mathsf{fold}(x))]$. Notice that the use of second projection is admissible due to the annotation of the first component $w \overset{0}{:} W$ — we are free to dispose of w. We need to show that $P[\mathsf{fst}(\mathsf{fold}(x))] = P[x]$ for every $x : W$, but as this is a type equality, unrestricted use of terms is permissible. The map $\mathsf{fst} : (w \overset{0}{:} W) \otimes P \xrightarrow{1} W$ is an $F_{A,B}$-algebra morphism, and thus the composite $\mathsf{fst} \circ \mathsf{fold} : W \xrightarrow{1} W$ is also one (*see diagram on the right*). Thus $\mathsf{fst} \circ \mathsf{fold} = \mathsf{id}$ holds by uniqueness of the mediating morphism out of W. The converse direction follows analogously. □



Inductively generated polynomial functors However, there is a caveat — most quantitative versions of standard data types are not quantitative containers in the above sense. Consider, for example, the natural numbers, the initial algebra of the polynomial functor $F(X) = \mathbf{1} + X$, or binary trees — the initial algebra of $G(X) = A + X \times X$. Their representations in "container normal form" crucially depend on isomorphisms $(\mathbf{0} \to X) \cong \mathbf{1}$ and $(\mathbf{Bool} \to X) \cong X \times X$, respectively, but their QTT counterparts do not hold: $(\mathbf{0} \xrightarrow{1} X) \ncong \mathbf{I}$ and $(\mathbf{Bool} \xrightarrow{1} X) \ncong X \otimes X$. Thus we resort to generating the class of *quantitative polynomial functors* inductively by:

$$F, G ::= \mathsf{Id} \mid \mathsf{Const}_A \mid F \underline{\otimes} G \mid F \underline{\oplus} G \mid F \,\underline{\&}\, G \mid A \underline{\xrightarrow{1}} X$$

Theorem 2 still holds for that class with the induction principle reformulated as:

$$\frac{w \overset{0}{:} W \vdash P \quad \vdash M : (w \overset{0}{:} F(W)) \otimes \widehat{F}_W(P, w) \to P(c(w))}{\vdash \mathsf{elim}(P, M) : (x \overset{1}{:} W) \to P[x]}$$

where $\widehat{F}_W$ is an appropriately defined predicate lifting $\widehat{F}_X : (P : X \to \mathsf{Type}) \to (F(X) \to \mathsf{Type})$. The proof can be carried out by using a distributive lemma for the dependent tensor and the predicate lifting:

Lemma 3. $F((w \overset{0}{:} W) \otimes P) \cong (w' \overset{0}{:} F(W)) \otimes \widehat{F}_W(P)$. □

Initial algebras of finitary polynomial functors (i.e., without the exponential $A \underline{\xrightarrow{1}} -$) can be constructed in the graph model [Atk18] based on linear realisability [Hos07]. In the future, we hope to extend this also to infinitary polynomial functors.

References

[AAG03] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. *Lecture Notes in Computer Science*, 2620:23–38, 2003.

[AGS17] Steve Awodey, Nicola Gambino, and Kristina Sojakova. Homotopy-initial algebras in type theory. *Journal of the ACM*, 63(6), 2017.

[Atk18] Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*, pages 56–65, New York, New York, USA, 2018. ACM Press.

[Ben95] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Lecture Notes in Computer Science*, volume 933, pages 121–135. Springer, Berlin, Heidelberg, 1995.

[CP02] Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. *Information and Computation*, 179(1):19–75, nov 2002.

[HJ98] Claudio Hermida and Bart Jacobs. Structural Induction and Coinduction in a Fibrational Setting. *Information and Computation*, 145(2):107–152, sep 1998.

[Hos07] Naohiko Hoshino. Linear Realizability. In *Computer Science Logic*, volume 4646 LNCS, pages 420–434. Springer Berlin Heidelberg, Berlin, Heidelberg, mar 2007.

[KPB15] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating Linear and Dependent Types. *ACM SIGPLAN Notices*, 50(1):17–30, may 2015.

[McB16] Conor McBride. I Got Plenty o' Nuttin'. In *Lecture Notes in Computer Science*, volume 9600, pages 207–233. Springer Verlag, 2016.

Interpreting Twisted Cubes as Partially Ordered Spaces

Gun Pinyo

University of Nottingham

Overview: *Twisted cubes* are a new kind of combinatorial shapes introduced in [9]. The idea of twisted cubes originates from an ambition to modify the cubes used in *cubical type theories* (e.g. [2, 3, 1]) in such a way that they are compatible with *directed type theories* (e.g. [5, 8, 10, 7]).

This abstract introduces a geometric representation of twisted cubes as *partially ordered spaces* (or *pospaces* for short), which are topological spaces each equipped with a *closed* partial order. In other words, this abstract "geometrically realises" twisted cubes from their combinatorial shapes to their (directed) geometric shapes, thus, they can be visualised in the same way as other geometric shapes for higher structures [6] such as simplices and (standard) cubes.

It turns out that the resulting pospace definition of twisted cubes can be quite similar to the pospace definition of standard cubes (which is also defined here for comparison purposes). In particular, the underlying topological spaces of both kinds of cubes are the same (up to linear transformation). This suggests the potential syntax of the "twisted cubical type theory" to be defined based on one of the cubical type theories [2, 3, 1] equipped with a further restriction associated with the partial orders of twisted cubes.

Partially Ordered Spaces: Topological spaces alone are not expressive enough to fully encode the representation of twisted cubes because paths in twisted cubes are not necessary reversible, therefore, our suitable definition of spaces must have some sense of direction.

Definition 1. A *partially ordered space* [4] is a topological space X equipped with a partial order $(\leqslant)$ on the set of points of X such that $\{\ (x,y) \in X^2 \ \mid\ x \leqslant y\ \}$ is a *closed set* in X^2.

Definition 2. A *dimap* $f : X \to Y$ between two pospaces is a continuous map between the underlying topological spaces that preserves the partial order, i.e. if $u \leqslant_X v$ then $f(u) \leqslant_Y f(v)$.

Example 3. For each dimension $n \in \mathbb{N}$, the Euclidean space $\mathbb{R}^n_{\mathsf{top}}$ can be upgraded to *the Euclidean pospace* $\mathbb{R}^n_{\mathsf{pospc}}$, where $(\vec{x} \leqslant_{\mathbb{R}^n_{\mathsf{pospc}}} \vec{y}) := \bigwedge_{i=0}^{n-1}(x_i \leqslant y_i)$ with notation $\vec{p} := (p_0, p_1, \ldots, p_{n-1})$.

Metric Spaces with Real-Valued Continuous Functions: Rather than manually defining pospaces, we will generate each of them based on a metric space (X, d) together with a continuous function $f : X \to \mathbb{R}_{\mathsf{top}}$.

Definition 4. $(\leqslant_f)$ is a relation on X such that $(x \leqslant_f y) := d(x,y) \leqslant (f(y) - f(x))$.

Lemma 5. Given the context above, the relation $(\leqslant_f)$ is a closed partial order.

Definition 6. Given the context above, $\mathsf{mkPospc}(X, f)$ is defined to be a pospace consisting of the topological space X, together with the partial order $(\leqslant_f)$ restricted to X.

Embedding Graphs to Pospaces: We require that our resulting pospaces will be compatible with $\square^n_{\mathsf{graph}}$ and $\bowtie^n_{\mathsf{graph}}$ which are defined to be the irreflexive version of graphs C_n and T_n in [9].

Definition 7. Given an *acyclic* graph $G = (V, E)$, we define $\mathsf{mkGraphPospc}(G)$ as a pospace (V^*, E^*) where V^* is the discrete space from V and E^* is the reflexive transitive closure of E.

Definition 8. Given an acyclic graph G and a pospace P, we define $\mathsf{canEmbed}(G, P)$ to be a proposition stating that there exists a dimap from $\mathsf{mkGraphPospc}(G)$ to a pospace P such that the underlying function is injective and contains every extreme point of P in its image.

Pospaces of Standard Cubes: $\mathsf{rank}^n_{\mathsf{std}}$ approximates $\mathbb{R}^n_{\mathsf{pospc}}$, thus, we define $\square^n_{\mathsf{pospc}}$ by it.

Definition 9. We define $\mathsf{rank}^n_{\mathsf{std}} : \mathbb{R}^n \to \mathbb{R}$ by setting $\mathsf{rank}^n_{\mathsf{std}}(\vec{x}) := \sum_{i=0}^{n-1} x_i$.

Lemma 10. $(x \leqslant_{\mathbb{R}^n_{\mathsf{pospc}}} y)$ implies $(x \leqslant_{\mathsf{rank}^n_{\mathsf{std}}} y)$, for all $x\ y \in \mathbb{R}^n$; and also vice versa if $n \leqslant 2$.

Definition 11. $\square^n_{\mathsf{top}}$, the topological space of *standard n-cube*, is defined to be a subspace of $\mathbb{R}^n_{\mathsf{top}}$ where $\vec{x} \in \square^n_{\mathsf{top}}$ iff $(0 \leqslant x_i \leqslant 1)$ i.e. $x_i \in [0,1]$ for all $i \in \mathbb{N}$ such that $i < n$.

Definition 12. $\square^n_{\mathsf{pospc}}$, the pospace of *standard n-cube*, is defined as $\mathsf{mkPospc}(\square^n_{\mathsf{top}}, \mathsf{rank}^n_{\mathsf{std}})$.

Theorem 13. The proposition $\mathsf{canEmbed}(\square^n_{\mathsf{graph}}, \square^n_{\mathsf{pospc}})$ holds where the underlying function $\mathsf{emb}^n_{\mathsf{std}} : \{0,1\}^n \to [0,1]^n$ is defined to be the inclusive from $\{0,1\}^n$ to $[0,1]^n$.

Pospaces of Twisted Cubes: The algorithm of twisted cubes that decides the directions of graph edges in $\bowtie^n_{\mathsf{graph}}$ involves the boolean operator "iff" (i.e. "not xor") on bits "0" and "1". This can be transformed to the multiplicative operator on real numbers -1 and 1, respectively.

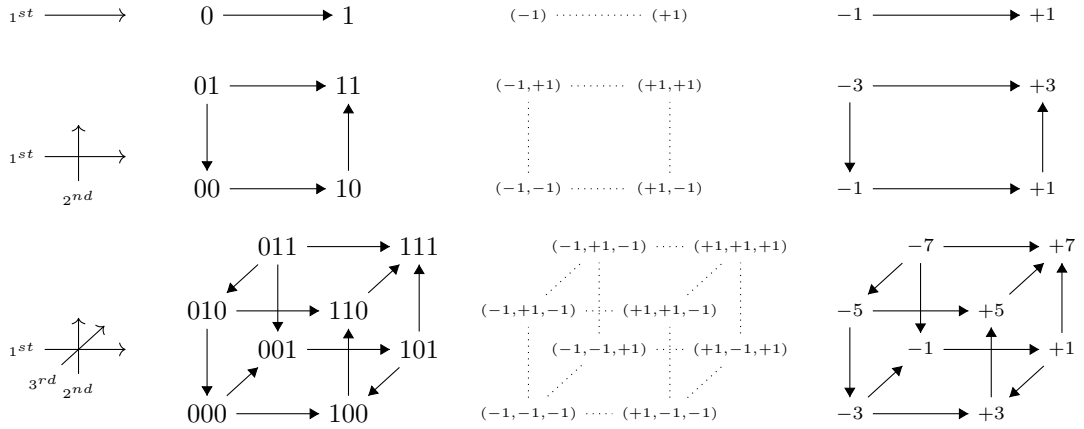
Definition 14. We define $\bowtie^n_{\mathsf{top}}$ to be the result of a dimension-wise linear transformation of $\square^n_{\mathsf{top}}$ by $((2 \cdot x_i) - 1)$, therefore, the interval of each dimension is stretched from $[0,1]$ to $[-1,1]$.

Definition 15. We define $\mathsf{rank}^n_{\mathsf{tw}} : \mathbb{R}^n \to \mathbb{R}$ as $\mathsf{rank}^n_{\mathsf{tw}}(\vec{x}) := \sum_{i=0}^{n-1}(x_i \cdot 2^{(n-1-i)} \cdot \prod_{j=0}^{i-1} x_j)$ or recursively (and equivalently) as $\mathsf{rank}^0_{\mathsf{tw}}() := 0$ and $\mathsf{rank}^{n+1}_{\mathsf{tw}}(x_0, \vec{x}) := x_0 \cdot (2^n + \mathsf{rank}^n_{\mathsf{tw}}(\vec{x}))$.

Definition 16. $\bowtie^n_{\mathsf{pospc}}$, the pospace of *twisted n-cube*, is defined as $\mathsf{mkPospc}(\bowtie^n_{\mathsf{top}}, \mathsf{rank}^n_{\mathsf{tw}})$.

Remark 17. Intuitively, $\mathsf{rank}^n_{\mathsf{tw}}$ is a modification of $\mathsf{rank}^n_{\mathsf{std}}$ where $(\prod_{j=0}^{i-1} x_j)$ is multiplied to x_i because each earlier dimension j could reverse the direction of dimension i. We also multiply $2^{(n-1-i)}$ to x_i in order to ensure that the sums of later terms will not outweigh the current term i.e. $\left|x_i \cdot 2^{-i} \cdot \prod_{j=0}^{i-1} x_j\right| > \sum_{k=i+1}^{n-1} \left|x_k \cdot 2^{-k} \cdot \prod_{j=0}^{k-1} x_j\right|$, which is effective because $\left|x_i\right| \leqslant 1$. Consequently, $\mathsf{rank}^n_{\mathsf{tw}}$ only works in $\bowtie^n_{\mathsf{top}}$; If we wants $\mathsf{rank}^n_{\mathsf{tw}}$ to work for the entire $\mathbb{R}^n_{\mathsf{top}}$, then we needs to change the codomain of $\mathsf{rank}^n_{\mathsf{tw}}$ to *hyperreal numbers* [11] and replace 2 with ω.

Example 18. The following diagram visualises Definition 15 by illustrating the anatomy of the twisted n-cube when $1 \leqslant n \leqslant 3$. The columns shows: (1.) the direction of each dimension. (2.) the graph $\bowtie^n_{\mathsf{graph}}$, (3.) the topological space $\bowtie^n_{\mathsf{top}}$, and (4.) the value of $\mathsf{rank}^n_{\mathsf{tw}}(p)$, for each extreme point p in $\bowtie^n_{\mathsf{pospc}}$, together with arrows in the direction that the value increases.



Theorem 19. The proposition $\mathsf{canEmbed}(\bowtie^n_{\mathsf{graph}}, \bowtie^n_{\mathsf{pospc}})$ holds where the underlying function $\mathsf{emb}^n_{\mathsf{tw}} : \{0,1\}^n \to [-1,1]^n$ is defined to be the composition between $\mathsf{emb}^n_{\mathsf{std}}$ in Theorem 13 and the dimension-wise linear transformation in Definition 14.

References

[1] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In Dan Ghica and Achim Jung, editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, 2014.

[3] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003.

[5] Daniel R Licata and Robert Harper. 2-dimensional directed type theory. *Electronic Notes in Theoretical Computer Science*, 276:263–289, 2011.

[6] nlab authors. geometric shape for higher structures, 2021. [Online; accessed 12-April-2021].

[7] Paige Randall North. Towards a directed homotopy type theory. *Electronic Notes in Theoretical Computer Science*, 347:223–239, 2019.

[8] Andreas Nuyts. Towards a directed homotopy type theory based on 4 kinds of variance. Master's thesis, KU Leuven, 2015.

[9] Gun Pinyo and Nicolai Kraus. From Cubes to Twisted Cubes via Graph Morphisms in Type Theory. In Marc Bezem and Assia Mahboubi, editors, *25th International Conference on Types for Proofs and Programs (TYPES 2019)*, volume 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[10] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories. *Higher Structures*, 1(1), 2017.

[11] A. Robinson. *Non-standard Analysis*. Princeton landmarks in mathematics and physics. Princeton University Press, 1974.

Novel rules of β-conversion in partial type theory

Jiří Raclavský[1] and Petr Kuchyňka[2]

[1] Masaryk University, Brno, the Czech Rep.
raclavsky@phil.muni.cz
[2] University of Defence, Brno, the Czech Rep.
p.kuchynka@gmail.com

Short abstract: *In partial type theory, i.e. a higher-order logical system that manipulates both total and partial functions, a precise formulation of valid rules of β-conversion and even their versions that substitute a value is possible if explicit substitution and two special evaluation terms are involved. We derive the latter rules from the primary versions of β-conversion rules and other primitive rules of the natural deduction for the system. In addition, we formulate and derive further novel variants of β-conversion rules which are also needed for capturing inferences with terms that denote partial functions.*

The pivotal rules of *type theory* (TT), i.e. a *higher-order logic* with a hierarchy of functions sorted in interpretations $\mathfrak{D}_\tau$ of *types* τ, are *rules of β-conversion* (i.e. *β-contraction*: $\vdash$; *β-expansion*: $\dashv$):

$$[\lambda\tilde{x}_m.C](\bar{D}_m) \dashv\vdash C_{(\bar{D}_m/\bar{x}_m)}$$

where $\tilde{X}_m$ is short for $X_1X_2...X_m$; $\bar{X}_m$ is short for $X_1, X_2, ..., X_m$; but $C_{(\bar{D}_m/\bar{x}_m)}$ is short for $C_{(D_1/x_1)...(D_m/x_m)}$ (where $C_{(D/x)}$ abbreviates $Sub(\ulcorner D\urcorner, \ulcorner x\urcorner, \ulcorner C\urcorner)$, see below).

However, within *partial* TT, i.e. a TT which embraces both total and partial functions,[1] the above classical formulation of β-contraction is not valid. For example,

$$[\lambda x.\lambda y. \div(x,x)](\div(3,0)) \not\longrightarrow_\beta \lambda y. \div(\div(3,0), \div(3,0)),$$

for $[\lambda x.\lambda y. \div(x,x)](\div(3,0))$ is *non-denoting* (because $D := \div(3,0)$ is non-denoting), but $\lambda y. \div(\div(3,0), \div(3,0))$ denotes a certain partial function. This is why Tichý [6], Moggi [4], Farmer [1] and others conditioned the rule by requiring that D entering β-reduction must be *denoting*.

In Tichý's [6] convenient *two-dimensional* (see e.g. Quieroz et al. [5], Tichý [7]) *natural deduction ND* for his *simple TT* (STT) with total and partial (multiargument) functions, his safe β-contraction rule[2] reads

$$(\beta\text{–CON}) \qquad \Gamma \Rightarrow [\lambda\tilde{x}_m.C](\bar{D}_m){:}\mathbf{a} \vdash \Gamma \Rightarrow C_{(\bar{D}_m/\bar{x}_m)}{:}\mathbf{a}$$

in which λ-terms are *'signed'* by :$\mathbf{a}$, which is a terse variant of $\cong \mathbf{a}$, where $\cong$ is a symbol of *congruence*; $\mathbf{a}$ is either a variable a or a constant a (Γ is a set of these so-called *matches* C:$\mathbf{a}$ and $\Rightarrow$ is a relation between sets of sequents and sequents of the form $\Gamma \Rightarrow C$:$\mathbf{a}$). In the left-hand part of the rule it requires the application written on the left-hand side of $\vdash$, and thus also its parts are *denoting*.

However, Tichý's proposal is too restrictive and it is one of the goals of the paper to remove this drawback. For example,

$$[\lambda x. \div(x,0)](3) \longrightarrow_\beta (\div(3,0))$$

[1] A *total / partial function-as-graph* maps all / some but not all members of its domain $\mathfrak{D}$ to some members of its codomain $\mathfrak{D}'$. Note that such functions differ from *functions-as-computations*.

[2] In this abstract, we omit β-expansion rule(s). Further, let C etc. be *typed* by types $\tau_{(i)}$ as follows: $C, \mathbf{a}/\tau; D_1, x_1/\tau_1; ...; D_m, x_m/\tau_m$, so $\lambda\tilde{x}_m.C/\langle\bar{\tau}_m\rangle \mapsto \tau$ (the type of functions from $\mathfrak{D}_{\tau_1} \times ... \times \mathfrak{D}_{\tau_m}$ to $\mathfrak{D}_\tau$), where $\bar{\tau}_m$ is short for $\tau_1, \tau_2, ..., \tau_m$. We assume the STT typing à Tichý [7] (no dependent types yet).

is not handled by (β-CON). To capture also such examples we propose

$$(\beta{-}\mathrm{CON}^{-}) \qquad \Gamma \Rightarrow [\lambda\tilde{x}_m.C](\bar{D}_m){:}\_; \Gamma \Rightarrow D_1{:}\mathbf{x}_1; ...; \Gamma \Rightarrow D_m{:}\mathbf{x}_m \vdash \Gamma \Rightarrow C_{(\bar{D}_m/\bar{x}_m)}{:}\_$$

where each $D_i{:}x_i$ says that D_i is denoting an object in the range of x_i, and $X{:}\_$ represents that X is *non-denoting*. (Note that 2D-inference proceeds on sequents, not on mere terms/formulae.)

Two key contributions of the present paper are:

1. *A formulation of yet unknown rules of β-conversion (e.g. ($\beta{-}\mathrm{CON}^{-}$)) for partial TT.*
2. *A derivation of the novel rules of β-conversion from primitive rules (e.g. (β-CON)) of the natural deduction ND for partial TT.*

Our system TT^* is not a ramified TT, cf. e.g. Kamareddine, Laan, Nederpelt [3], though Tichý's late TT utilises *ramification* of his early TT. It is metalogical in its spirit and the respective ND allows derivation results such as those mentioned in point 2.

Most of our effort pertains to a logically satisfactory development of the following two key contributions inbuilt in TT^*:

3. *An extension of the partial TT by 'evaluation terms', namely capture(s) $\ulcorner C \urcorner$ (whose denotation is C as such) and immersion(s) $[\![C]\!]_\tau$ (whose denotation is C's denotation, if any), and a proposal of ND rules for these terms.*[3]
4. *An appropriate extension of the substitution function Sub and a modification of rules using explicit substitution.*

The 'evaluation terms' allows us to derive our further novel rules of β-contraction:

$$(\beta{-}\mathrm{CON}^{V}) \qquad \Gamma \Rightarrow [\lambda\tilde{x}_m.C](\bar{D}_m){:}\mathbf{a} \vdash \Gamma \Rightarrow C_{\ulcorner(\bar{D}_m/\bar{x}_m)\urcorner}{:}\mathbf{a}$$

$$(\beta{-}\mathrm{CON}^{V-}) \qquad \Gamma \Rightarrow [\lambda\tilde{x}_m.C](\bar{D}_m){:}\_ \vdash \Gamma \Rightarrow C_{\ulcorner(\bar{D}_m/\bar{x}_m)\urcorner}{:}\_$$

in which V indicates that one substitutes the *value* of D and $\ulcorner(\bar{D}_m/\bar{x}_m)\urcorner$ is a derived variant of $(\bar{D}_m/\bar{x}_m)$ that substitutes the value of (each) D_i, not D_i as such (cf. also below).

A motivation for such novel rules provide pieces of inference such as e.g. (let *Succ* denote the familiar successor function-as-graph); its formalisation is right below:

$$\frac{\begin{array}{c}\textit{Succ(2) is such that the result of substitution of its value}\\ \textit{for } n \textit{ in } n \div 0 \textit{ is (congruent with) } \_ \textit{ (i.e. nothing at all).}\end{array}}{\textit{The value of the result of substitution of Succ(2)'s value for } n \textit{ in } n \div 0 \textit{ is } \_.}$$

$$\frac{[\lambda n'.Sub(\ulcorner(n')\urcorner, \ulcorner n\urcorner, \ulcorner \div(n,0)\urcorner)](Succ(2)){:}\_}{[\![Sub(\ulcorner(Succ(2))\urcorner, \ulcorner n\urcorner, \ulcorner \div(n,0)\urcorner)]\!]_\tau{:}\_}\ (\beta\text{-}\mathrm{CON}^{V-})$$

Our approach can therefore capture that

a. one substitutes something in the computation $\div(n,0)$ as such (hence $\ulcorner \div(n,0)\urcorner$), not to its (non-existent) result
b. one substitutes the value of the computation $Succ(2)$, not $Succ(2)$ as such (hence one uses not $\ulcorner Succ(2)\urcorner$, but $\ulcorner(Succ(2))\urcorner$, where $\ulcorner(.)\urcorner$ is a function that delivers the constant that stands for the semantic value of a term and e.g. n' is free in $\ulcorner(n')\urcorner$)
c. it is the value of the result of substitution (hence $[\![...]\!]_\tau$), not the computation as such, which is compared by $\cong$ with $\_$
d. it is our novel and derived rule (β-CON^{V-}) not any other β-reduction rule, which is appropriate for obtaining the conclusion from the premiss.

[3] See e.g. Farmer [2], but we rather follow Tichý [7] and significantly expand his system.

References

[1] William M. Farmer. "A Partial Functions Version of Church's Simple Theory of Types". In: *Journal of Symbolic Logic* 55.3 (1990), pp. 1269–1291. DOI: https://doi.org/10.2307/2274487.

[2] William M. Farmer. "Incorporating Quotation and Evaluation into Church's Type Theory: Syntax and Semantics". In: *Intelligent Computer Mathematics. CICM 2016. Lecture Notes in Computer Science, vol 9791*. Ed. by B. Miller B. L. de Moura F. Tompa M. Kohlhase M. Johansson. Springer, 1989, pp. 83–98. DOI: https://doi.org/10.1007/978-3-319-42547-4_7.

[3] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory. From Its Origins until Today*. Springer, 2004. ISBN: 978-1402023347.

[4] Eugen Moggi. "The Partial Lambda-Calculus". Ph.D. thesis. University of Edinburgh, 1988.

[5] Ruy J G B de Queiroz, Anjolina G de Oliveira, and Dov M Gabbay. *The Functional Interpretation of Logical Deduction*. World Scientific, 2011. ISBN: 978-981-4360-95-1.

[6] Pavel Tichý. "Foundations of Partial Type Theory". In: *Reports on Mathematical Logic* 14 (1982), pp. 57–72.

[7] Pavel Tichý. *The Foundations of Frege's Logic*. Walter de Gruyter, 1988. ISBN: 978-3110116687.

Eliminating Infinitary Induction-induction

Filippo Sestini and Thorsten Altenkirch

School of Computer Science, University of Nottingham, UK

In the study of the metatheory of type theory, one is often interested in finding ways to reduce a given class of inductive types to another. One reason is that it furthers our understanding of the class of types being reduced. Another, more practical reason is that from such reductions we can often extract ways to extend the capabilities of existing proof-assistants based on type theory beyond their original design. Inductive-inductive types [5] (IITs) are a particularly evident example, since several well-known and widely-used proof assistants like Coq or Lean do not directly support them. Finitary [1] induction-induction can be reduced to inductive families [4,6], which means that these proof-assistants could be made to support it (for example, via metaprogramming) without the need to change the core foundational theory underlying them.

Finitary IITs are a rich subclass of induction-induction, that even includes the internal representation of type theory in type theory [2]. Still, there are interesting examples of IITs that are not finitary [1]. Unfortunately the reduction method for finitary IITs cannot be applied out-of-the-box to infinitary types unless we assume function extensionality. This paper presents work towards a reduction from infinitary IITs to inductive families that does not require *funext*. This is important in particular when we want to use the reduction when eliminating function extensionality as in [1]. We show that a modified version of the reduction method in [6] works on some infinitary IITs, by deriving signatures, constructors, and eliminators for a concrete infinitary IIT in a metatheory without induction-induction or *funext*. We finally discuss our plans to prove this reduction in the general case for a wide subclass of infinitary IITs.

Metatheory We work in standard MLTT extended with a universe of strict propositions **Prop**, like implemented in Agda or Coq [3], and a **Prop**-valued identity type $\mathsf{Id}_p : \{A : \mathbf{Type}\} \to A \to A \to \mathbf{Prop}$ equipped with a *strong transport* combinator $\mathsf{transp} : \forall\{A\,a_0\,a_1\}(C : A \to \mathbf{Type}) \to \mathsf{Id}_p\ a_0\ a_1 \to C\ a_0 \to C\ a_1$. Note that this combinator allows to transport over a strict-propositional equation along *proof-relevant* types, so it's a non-trivial extension to the metatheory that doesn't directly follow from Id_p's inductive definition. Nevertheless, unlike function extensionality this principle has an obvious computational interpretation in an intensional setting, even though its metatheory hasn't yet been fully established.

Reduction method We now illustrate the main ideas of the reduction, which is based on the method described in [6] for finitary IITs, but with crucial differences that will be made clear throughout. We consider as a running example the type of contexts $\mathsf{Con} : \mathbf{Type}$ and types indexed by contexts $\mathsf{Ty} : \mathsf{Con} \to \mathbf{Type}$. Note that the constructor for Π-types has been altered to be infinitary.

$$\bullet : \mathsf{Con} \qquad \iota : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma$$

$$-,- : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Con} \qquad \pi : \{\Gamma : \mathsf{Con}\}(A : \mathsf{Ty}\ \Gamma) \to (\mathsf{N} \to \mathsf{Ty}\ (\Gamma, A)) \to \mathsf{Ty}\ \Gamma$$

$$\mathsf{elim}_{\mathsf{Con}} : (\Gamma : \mathsf{Con}) \to \mathsf{Con}^D\ \Gamma \qquad \mathsf{elim}_{\mathsf{Ty}} : \forall\{\Gamma\}(A : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}^D\ \Gamma\ A\ (\mathsf{elim}_{\mathsf{Con}}\ \Gamma)$$

[1] We say that an IIT is finitary, if it defines finitely branching trees, i.e. there are no Π-types that targets any of the types currently defined in its codomain. Otherwise it is infinitary.

where $\mathsf{Con}^D : \mathsf{Con} \to \mathbf{Type}, \mathsf{Ty}^D : \forall\{\Gamma\} \to \mathsf{Con}^D\,\Gamma \to \mathsf{Ty}\,\Gamma \to \mathbf{Type}$ are components of any *displayed algebra* of $\mathsf{Con}, \mathsf{Ty}$. The objective is to encode this IIT and its elimination principle in our metatheory, which doesn't support induction-induction. The idea is to define a pair of types $\mathsf{Con}_0, \mathsf{Ty}_0 : \mathbf{Type}$, obtained from $\mathsf{Con}, \mathsf{Ty}$ by "erasing" all indices, and well-typing predicates $\mathsf{Con}_1 : \mathsf{Con}_0 \to \mathbf{Prop}, \mathsf{Ty}_1 : \mathsf{Con}_0 \to \mathsf{Ty}_0 \to \mathbf{Prop}$ that restore the information lost in the erasure. Unlike in [6], we define the predicates as strict propositions. This is to avoid proving propositionality by induction, which would require function extensionality in general. The constructors of erased types and predicates are derived just like in the finitary case. We write the cases for π^∞ below:

$$\pi_0^\infty : \mathsf{Con}_0 \to \mathsf{Ty}_0 \to (\mathsf{N} \to \mathsf{Ty}_0) \to \mathsf{Ty}_0$$
$$\pi_1^\infty : \forall\{\Gamma_0\,A_0\,B_0\} \to \mathsf{Con}_1\,\Gamma_0 \to \mathsf{Ty}_1\,\Gamma_0\,A_0 \to ((n : \mathsf{N}) \to \mathsf{Ty}_1\,(\Gamma_0, A_0)\,(B_0\,n)) \to \mathsf{Ty}_1\,\Gamma_0\,(\pi_0^\infty\,\Gamma_0\,A_0\,B_0)$$

We can recover the original IIT as $\mathsf{Con} :\equiv \Sigma\ (\Gamma_0 : \mathsf{Con}_0)\ (\mathsf{Con}_1\ \Gamma_0)$ and $\mathsf{Ty}\ \Gamma :\equiv \Sigma\ (A_0 : \mathsf{Ty}_0)\ (\mathsf{Ty}_1\ (\pi_1\ \Gamma)\ A_0)$. Recovering the constructors is a straightforward pairing.

Since we can't rely on induction-induction (or rather recursion-recursion), we won't define the eliminators directly, but instead inductively define their graphs as relations. We then show by induction on the erased types $\mathsf{Con}_0, \mathsf{Ty}_0$ that they are total.

$$\mathsf{data}\ \mathsf{R\text{-}Con} : (\Gamma : \mathsf{Con}) \to \mathsf{Con}^D\ \Gamma \to \mathbf{Type}$$
$$\mathsf{data}\ \mathsf{R\text{-}Ty}\ \ : \{\Gamma : \mathsf{Con}\}(A : \mathsf{Ty}\ \Gamma)(\gamma : \mathsf{Con}^D\ \Gamma) \to \mathsf{Ty}^D\ \Gamma\ A\ \gamma \to \mathbf{Type}$$
$$\mathsf{existsCon} : \forall\,\Gamma \to \Sigma(\Gamma^D : \mathsf{Con}^D\,\Gamma)\,(\mathsf{R\text{-}Con}\,\Gamma\,\Gamma^D)$$
$$\mathsf{existsTy}\ \ : \forall\,\{\Gamma\,\Gamma^D\}\,(A : \mathsf{Ty}\,\Gamma) \to \mathsf{R\text{-}Con}\,\Gamma\,\Gamma^D \to \Sigma(A^D : \mathsf{Ty}^D\,\Gamma^D\,A)(\mathsf{R\text{-}Ty}\,A\,\Gamma^D\,A^D)$$

Note that unlike in [6], we prove totality but not right-uniqueness, as the latter requires function extensionality. The way $\mathsf{existsTy}$ is defined is crucial in avoiding the need for right-uniqueness. In fact, at first glance we could have equivalently defined it to return $\Gamma^D : \mathsf{Con}^D\,\Gamma$ and $\mathsf{R\text{-}Con}\,\Gamma\,\Gamma^D$ as an output along A^D, rather than requiring it as an input. However, the first choice would have forced us to prove right-uniqueness.

The strong transport rule is essential to define $\mathsf{existsTy}$. In many cases, we would need to pattern match on the well-typing component of A to expose equational constraints on Γ. However, in our case this has become impossible, since Ty_1 is a strict proposition. We solve this by expressing these equations in terms of Id_p, then rely on strong transport to use them.

The eliminators can finally be defined immediately from first projections:
$\mathsf{elim}_{\mathsf{Con}}\,\Gamma :\equiv \pi_1(\mathsf{existsCon}\,\Gamma), \mathsf{elim}_{\mathsf{Ty}}\,\{\Gamma\}\,A :\equiv \pi_1(\mathsf{existsTy}\,A\,(\pi_2(\mathsf{existsCon}\,\Gamma)))$.

Towards a general reduction method We have seen this reduction method on an example. We are currently working on proving that it works in general, on a subclass of all infinitary IITs. This subclass has been characterized considering technical limitations of the reduction method, as well as simplicity.

First of all, we restricted our scope to non-indexed infinitary IITs with two fixed sorts of the form $A : \mathsf{Type}$ and $B : A \to \mathsf{Type}$. The number of sorts doesn't seem to be particularly important, as we should be able to equivalently express any IIT using only two sorts (although this fact seems to be folklore.) In the future we certainly want to extend the method to indexed IITs once it is proved to work for non-indexed ones.

One more serious limitation that we are seemingly forced to impose is linearity of the variables that appear in the conclusion type of constructors for B. For example, if $\mathsf{ctr}_A : A \to A \to A$ is a constructor for A, we would allow $\mathsf{ctr}_B : (a_0 : A)(a_1 : A) \to B\,(\mathsf{ctr}_A\,a_0\,a_1)$ but not $\mathsf{ctr}'_B : (a : A) \to B\,(\mathsf{ctr}_A\,a\,a)$. This is because the presence of non-linear variables forces us to prove right-uniqueness of the relations. However, we believe that this can be addressed by expressing non-linearity using a linear equality relation inductively defined alongside A and B.

References

[1] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, Christian Sattler, and Filippo Sestini. Constructing a universe for the setoid model. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures*, pages 1–21, Cham, 2021. Springer International Publishing.

[2] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 16, page 1829, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi.org/10.1145/2837614.2837638, doi:10.1145/2837614.2837638.

[3] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages*, pages 1–28, January 2019. URL: https://hal.inria.fr/hal-01859964, doi:10.1145/329031610.1145/3290316.

[4] Ambrus Kaposi, András Kovács, and Ambroise Lafont. For finitary induction-induction, induction is enough. In Marc Bezem and Assia Mahboubi, editors, *25th International Conference on Types for Proofs and Programs (TYPES 2019)*, volume 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13070, doi:10.4230/LIPIcs.TYPES.2019.6.

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

[6] Jakob von Raumer. *Higher Inductive Types, Inductive Families, and Inductive-Inductive Types*. PhD thesis, University of Nottingham, 2019.

Modelling Smart Contracts of Bitcoin in Agda

Anton Setzer[1]*and Bogdan Gabriel Lazar[1]†

Dept. of Computer Science, Swansea University, Swansea, UK

Abstract

This work is based on the first author's model of the transaction structure of Bitcoin using inductive-recursive data types in the theorem prover Agda. We extend this model by adding smart contracts written in the byte code language Script of Bitcoin. The goal is to use it for verifying the correctness of smart contracts. The use of non-local conditional instructions is solved without the use of jump addresses.

Cryptocurrencies are currently widely discussed. Many cryptocurrencies support smart contracts, a feature with big potential. Smart contracts are programs stored on the blockchain which are automatically executed when conditions are met, resulting in changes of data storage and monetary transactions. They are increasingly used for non-monetary applications. There is no legal framework at present to prevent the malicious execution of a smart contract. This contrasts with ordinary contracts where a legal framework exists, which void certain malicious uses of clauses written in a contract. Due to the absence of legal protection, smart contracts require a much higher level of correctness. Together with the fact that smart contracts are small and therefore easy to manage, and because mistakes might have substantial monetary consequences, smart contracts are a prime target for machine checked correctness proofs.

Bitcoin's language for smart contracts is the byte code language Script.[1] It is a Forth-like stack machine (Sect. 6 of [Ant17]). The instructions manipulate the stack, the only memory available. They may as well result in abortion of the program. In order to check signatures, Script can refer to a message extracted from the current transaction. Smart contracts might be executed after a certain amount of time and can therefore refer to the current time. Script does not have jumps; therefore, programs will always terminate. It has control flow statements, consisting of operations OP_IF/OP_NOTIF, OP_ELSE, OP_ENDIF. Depending on the top element of the stack after an OP_(NOT)IF the if-case or else-case (if it exists) is executed. This avoids miscalculation with forward jumps, a widespread problem in assembly languages.[2]

In [Set18, Set19] (see as well the PhD thesis [dS20]) the first author introduced a model of the transaction dag of Bitcoin, based on an inductive-recursive data structure. The transactions were defined inductively while recursively computing the set of unspent transaction outputs. Transactions consist of a list of inputs and a list of outputs. An input consists of a previous unspent transaction output (utxo), a public key, and a signature. An output consists of an amount, and an address. An input is correct if the public key hashes to the address. Furthermore, the signature needs to sign the relevant part of the message using the private key corresponding to the public key. In addition, there are global conditions for transactions such as that the sum of outputs needs to be less than the sum of inputs.

*a.g.setzer@swansea.ac.uk, http://www.cs.swan.ac.uk/~csetzer/

†lazarbogdan90@yahoo.com

[1]The cryptocurrency Ethereum has a high-level language Solidity for smart contracts, which on the blockchain is first compiled into the byte code of the Ethereum Virtual machine EVM. The EVM shares great similarities with Script. An attacker can directly target the EVM, therefore verification of smart contracts in Ethereum needs to address the byte code level. In this talk we will focus on Script, but techniques used there can be adapted to verify programs of the EVM.

[2]In contrast the EVM has arbitrary jumps, has reference to external memory, and replaces the conditionals of Script by conditional jumps.

In order to extend this model to include Script, the public key and signature in the input are replaced by an input script, called scriptSig. The address of the output is replaced by an output script, called scriptPubKey. To verify an input, we execute, starting with the empty stack, first the scriptSig followed by the scriptPubKey. The input is correct, if the execution is not aborted, at the end the stack is nonempty, and the top element of the stack is not false. To prevent an open OP_IF operation from the scriptSig to affect scriptPubKey, we will add our own separating instruction in between to make sure that the two parts are executed separately[3].

In order to carry out this verification we introduce an interpreter for Script in Agda. There are several groups of instructions. The first group of instructions are stack manipulating instructions. They assume a certain number of elements on the stack and replace them by different elements. For instance, OP_ADD assumes two elements on the stack, and replaces them by their sum. If there are not enough elements on the stack or certain conditions are not fulfilled the instructions abort (e.g., in case of OP_VERIFY if the top element is not true). Considering a possible failure all these instructions translate into functions of type

$$\text{Stack} \to \text{Maybe Stack}$$

The second group of instructions are those referring to the message representing the part of the instruction to be signed, or the time. They translate into functions

$$\text{Time} \to \text{Msg} \to \text{Stack} \to \text{Maybe Stack}$$

For dealing with conditionals, we need a second stack which gives information about the nesting of conditionals, and whether the current if- or else-case is to be executed. The elements are ifCase and elseCase, corresponding to the situation where we are in the if- or else-case of a conditional to be executed; ifSkip and elseSkip, where we are in an if- or else-case not to be executed; and ifIgnore, corresponding to the situation where we have a complete if-then-else which occurs inside an if- or else-case which is to be skipped. Note that we didn't introduce any jump instructions which would make verification more difficult. The operations correspond to functions of type

$$(\text{Stack} \times \text{IfStack}) \to \text{Maybe } (\text{Stack} \times \text{IfStack})$$

After lifting these different functions (using a higher order function these liftings can be done easily in a generic way), we obtain operations of type

$$\text{Time} \to \text{Msg} \to (\text{Stack} \times \text{IfStack}) \to \text{Maybe } (\text{Stack} \times \text{IfStack})$$

This lifting needs to take care of the fact that in case of ifSkip, elseSkip, ifIgnore on top of the IfStack, all non-conditional instructions need to be ignored.

As it stands the model is currently a prototype, since only a part of the language for smart contracts has been added. Once it is complete it needs to be thoroughly tested relative to Bitcoin Core. Ideally one would rewrite Bitcoin core in Agda, based e.g., on Haskoin Core [hac21]. The next step would be to use this approach to verify smart contracts, where the challenge is to specify what it means for a smart contract to be correct, which involves temporal features. Solving this challenge would allow to give more abstract specifications of smart contracts expressing directly its correctness rather than just looking for possible attacks or showing that a contract is bisimilar to an abstract smart contract assumed to be contract – these are the techniques commonly used for specifying the correctness of smart contracts. This would also allow to verify more generic forms of smart contracts, something which we don't assume is possible using the normally used automated theorem techniques. Another challenge is to expand the use of smart contracts to a language like the EVM or to directly add object-based programming to the language, which could make use of our work on integrating object-based programming into Agda. [AAS17, Set07].

[3] After suspecting a problem when developing the specification, we learned that there was indeed originally a problem in Bitcoin which was fixed in 2010, see Sect. 6 of [Ant17].

References

[AAS17] Andreas Abel, Stephan Adelsberger, and Anton Setzer. Interactive programming in Agda – Objects and graphical user interfaces. *Journal of Functional Programming*, 27, Jan 2017. https://doi.org/10.1017/S0956796816000319.

[Ant17] Andreas M Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly, 2nd edition, 2017.

[dS20] Guilherme Horte Alvares da Silva. *A simplified version of Bitcoin, implemented in Agda*. PhD thesis, Escola de Matemática Aplicada, FGV, Brasil, 2020. Available from http://bibliotecadigital.fgv.br/dspace/bitstream/handle/10438/29110/thesis.pdf?sequence=1&isAllowed=y.

[hac21] hackage.haskell.org. haskoin-core: Bitcoin & Bitcoin Cash library for Haskell. Vers 0.20.3, 17 May 2021. https://hackage.haskell.org/package/haskoin-core.

[Set07] Anton Setzer. Object-oriented programming in dependent type theory. In Henrik Nilsson, editor, *Trends in Functional Programming Volume 7*, pages 91 – 108, Bristol and Chicago, 2007. Intellect.

[Set18] Anton Setzer. Modelling Bitcoin in Agda. *arXiv*, arXiv:1804.06398:27, 17 April 2018. https://arxiv.org/abs/1804.06398.

[Set19] Anton Setzer. A model of the blockchain using induction-recursion. In Mark Bezem and Niels van der Weide, editors, *TYPES 2019. 25th International Conferences on Types of Proofs and Programs. Abstracts*, pages 102 – 103, Oslo, Norway, 2019. Centre for Advanced Study (CAS), The Norwegian Academy of Science and Letters. Available from https://eutypes.cs.ru.nl/eutypes_pmwiki/uploads/Main/books-of-abstracts-TYPES2019.pdf.

Size-Based Termination for Non-Positive Types

Yuta Takahashi*

IHPST (UMR 8590), Université Paris 1 Panthéon-Sorbonne, CNRS, Paris, France
yuuta.taka84@gmail.com

Background. Since the works [10, 15], several typed λ-calculus systems were combined with user-defined rewrite rules, and the termination (i.e. strong normalisation) problem of the combined systems was discussed: for instance, simply typed λ-calculus [9, 11, 13, 7], polymorphic λ-calculus [10, 15, 12], $\lambda\Pi$-calculus [8], the Calculus of Constructions [18, 5, 6], λ-cube [2], pure type systems [3, 4]. Rewrite rules can make these systems more expressive and efficient, and a termination criterion for combined systems provides a sufficient condition for the termination of the rewrite relation (i.e. the reduction relation) in a given combined system. Of course, there are some non-terminating and interesting combined systems, but here we are interested in terminating systems only.

The performance of a combined system depends on not only its type discipline but also the range of rewrite rules whose termination is guaranteed. For instance, while Jouannaud-Okada's work [12] handles polymorphic λ-calculus and Blanqui-Jouannaud-Okada's work [9] does not, the termination criterion in the latter shows the termination of the recursion principle for the Brouwer ordinal type, which cannot be shown by the criterion in the former. The Brouwer ordinal type is a type of well founded trees and a typical example of strictly positive inductive types. Later, Blanqui ([7]) extended the criterion in [9] so that the termination of some rewrite rules for non-strictly positive inductive types is guaranteed. Though the setting of [9, 7] is simply typed λ-calculus, their termination criteria are powerful enough.

Aim. We reinforce the termination criterion in [7] further by making it possible to verify the termination of some rewrite rules on types which are called *non-positive types*. When we denote arrow types by $T \Rightarrow U$, a non-positive type means a sort (i.e. a basic type) B with a constructor $\mathsf{c} : T_1 \Rightarrow \cdots \Rightarrow T_n \Rightarrow \mathsf{B}$ such that B occurs in some T_i negatively. As shown in [14, 16, 5], there are some non-positive types such that recursion principles for them induce non-termination. This indicates the difficulty in finding a terminating example of recursion principles for non-positive types. However, if one considers rewrite rules which are different from recursion principles, one can think of some rewrite rules on non-positive types whose termination is guaranteed. It is desirable to extend the criterion in [7] in this respect.

Approach. The approach of [7] to a termination criterion uses computability predicates with size annotations. Roughly speaking, its termination criterion is formulated in the following way: first, a computability predicate is assigned to each type T by extending an interpretation $\mathbb{I}$ of sorts. For any sort B, $\mathbb{I}(\mathsf{B})$ is a computability predicate annotated by ordinals as sizes: $\mathbb{I}(\mathsf{B})$ is equal to $\sup\{\mathcal{S}^{\mathsf{B}}_{\mathfrak{a}} \mid \mathfrak{a} < \mathfrak{h}\}$ for some limit ordinal $\mathfrak{h}$ and some ordinal-indexed family $(\mathcal{S}^{\mathsf{B}}_{\mathfrak{a}})_{\mathfrak{a}<\mathfrak{h}}$ of computability predicates, where $\mathcal{S}^{\mathsf{B}}_{\mathfrak{a}} \subseteq \mathcal{S}^{\mathsf{B}}_{\mathfrak{b}}$ holds for any $\mathfrak{a}, \mathfrak{b}$ with $\mathfrak{a} \leq \mathfrak{b}$. This kind of ordinal-indexed family of computability predicates is called a *stratification*. The interpretation $\mathbb{I}$ is extended to all types by defining $\mathbb{I}^*(\mathsf{B}) := \mathbb{I}(\mathsf{B})$ and

$$\mathbb{I}^*(T \Rightarrow U) := \mathbb{I}^*(T) \Rightarrow^* \mathbb{I}^*(U) := \{t \mid ts \in \mathbb{I}^*(U) \text{ for any } s \in \mathbb{I}^*(T)\}.$$

*Supported by JSPS (Japan Society for the Promotion of Science) Overseas Research Fellowship. We thank Frédéric Blanqui for valuable remarks and discussions on size-based termination.

Finally, it is shown that if a given rewrite system satisfies the termination criterion, then any term t of type T belongs to $\mathbb{I}^*(T)$; this implies that t is terminating.

In the approach above, an ordinal is assigned to each term in $\mathbb{I}(\mathsf{B})$ as its size. For instance, consider the Brouwer ordinal type O again: O has the three constructors

$$\mathsf{zero} : \mathsf{O} \qquad \mathsf{succ} : \mathsf{O} \Rightarrow \mathsf{O} \qquad \mathsf{lim} : (\mathsf{N} \Rightarrow \mathsf{O}) \Rightarrow \mathsf{O}$$

where N is the natural number type. Let $\mathbb{I}(\mathsf{O}) = \sup\{\mathcal{S}^{\mathsf{O}}_{\mathfrak{a}} \mid \mathfrak{a} < \mathfrak{h}\}$ be the case, then a typical case is the following: when a term t is of the normal form with $t \in \mathbb{I}(\mathsf{N}) \Rightarrow^* \mathcal{S}^{\mathsf{O}}_{\mathfrak{a}}$, then we have $\mathsf{lim}\, t \in \mathcal{S}^{\mathsf{O}}_{\mathfrak{a}+1}$. This is in particular useful for handling recursive calls of functions on inductive types, since we can consider that the recursive call of f in the definition of a function $\mathsf{f}(\mathsf{lim}\, t)$ is "smaller than" $\mathsf{f}(\mathsf{lim}\, t)$ thanks to size annotation for $\mathsf{lim}\, t$.

The main obstacle in extending this method of [7] to non-positive types is as follows: let $[\mathsf{B} : \mathcal{X}, \mathbb{I}]^*T$ be the interpretation of T obtained from $\mathbb{I}_1$ such that for any sort C, $\mathbb{I}_1(\mathsf{C}) = \mathcal{X}$ holds if $\mathsf{C} = \mathsf{B}$, otherwise $\mathbb{I}_1(\mathsf{C}) = \mathbb{I}(\mathsf{C})$. Then, a crucial fact in the method of [7] is that if B occurs in T only positively, then $[\mathsf{B} : \mathcal{X}_1, \mathbb{I}]^*T \subseteq [\mathsf{B} : \mathcal{X}_2, \mathbb{I}]^*T$ holds whenever $\mathcal{X}_1 \subseteq \mathcal{X}_2$ holds. This monotonicity property enables one to define a stratification $\mathcal{S}_0 \subseteq \mathcal{S}_1 \subseteq \cdots \subseteq \mathcal{S}_{\mathfrak{a}} \subseteq \cdots$ in the bottom-up way, but this property does not hold if B occurs in T negatively.

We remove this obstacle by utilising the inflationary fixed-point construction ([17]), which does not assume the monotonicity of operators for fixed points as explained in [1]. This construction provides the following obvious monotonicity to any ordinal-indexed family $(\mathcal{S}_{\mathfrak{c}})_{\mathfrak{c}<\mathfrak{h}}$ of computability predicates: if $\mathfrak{a} \leq \mathfrak{b}$ holds then $\bigcup_{\mathfrak{c}\leq\mathfrak{a}}([\mathsf{B} : \mathcal{S}_{\mathfrak{c}}, \mathbb{I}]^*T) \subseteq \bigcup_{\mathfrak{c}\leq\mathfrak{b}}([\mathsf{B} : \mathcal{S}_{\mathfrak{c}}, \mathbb{I}]^*T)$ holds, where B may occur in T negatively. A trade-off is that, for non-positive types, we need to reformulate a size-based termination argument only with pre-fixed points.

Our construction of computability predicates for non-positive types enables us to extend the accessibility property. Roughly speaking, the definition of the accessibility property in [7] says that a term t is accessible in a term s if there are terms $\mathsf{c}_1 s^1_1 \cdots s^1_{k_1}, \ldots, \mathsf{c}_n s^n_1 \cdots s^n_{k_n}$ such that the type of each $\mathsf{c}_m s^m_1 \cdots s^m_{k_m}$ is a sort (i.e. each constructor c_m is fully applied), and

1. $s = \mathsf{c}_1 s^1_1 \cdots s^1_{k_1}$,
2. $t = s^n_i$ holds for some i, and the type B_n of $\mathsf{c}_n s^n_1 \cdots s^n_{k_n}$ occurs in the type of t only positively (i.e. t is a positive argument of c_n),
3. for any m with $2 \leq m \leq n$, $\mathsf{c}_m s^m_1 \cdots s^m_{k_m} = s^{m-1}_i$ holds for some i.

We extend the clause 2. of this property so that B_n may occur in the type of t negatively (i.e. B_n can be a non-positive type). Then, Accessibility condition of our termination criterion is as follows: if a variable x occurs in the right-hand side r of a rewrite rule $\mathsf{f} l_1 \cdots l_n \to r$ with a function symbol f, then either $x = l_i$ holds or x is accessible in l_i for some i.

Our termination criterion guarantees the termination of the following rewrite system $\mathcal{R}$, though it is a toy example: let B be a sort with the constructors $\mathsf{c}_1 : \mathsf{B}$, $\mathsf{c}_2 : (\mathsf{B} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{B}$ and $\mathsf{c}_3 : (\mathsf{B} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{B}$. Then, $\mathcal{R}$ consists of the rules (1) $\mathsf{f}\mathsf{c}_1 \to \mathsf{c}_1$, (2) $\mathsf{f}(\mathsf{c}_2 a) \to \mathsf{c}_3 a$ and (3) $\mathsf{f}(\mathsf{c}_3 a) \to \mathsf{c}_2 a$ with $\mathsf{f} : \mathsf{B} \Rightarrow \mathsf{B}$. Note that $\mathcal{R}$ does not satisfy Accessibility condition in the sense of [7], because the right-hand sides of the second and third rules include a variable a whose type has a negative occurrence of B.

In sum, we extend the termination criterion in [7] to non-positive types by size-based termination with the inflationary fixed-point construction. A future research direction is to formulate our termination criterion for some dependent type system, and then find an interesting example of rewrite systems on non-positive types whose termination is guaranteed by our criterion.

References

[1] Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012*, pages 1–11, 2012.

[2] Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic-lambda-cube. *J. Funct. Program.*, 7(6):613–660, 1997.

[3] Gilles Barthe and Herman Geuvers. Modular properties of algebraic type systems. In *Higher-Order Algebra, Logic, and Term Rewriting, Second International Workshop, HOA '95, Paderborn, Germany, September 21-22, 1995, Selected Papers*, pages 37–56, 1995.

[4] Gilles Barthe and Femke van Raamsdonk. Termination of algebraic type systems: The syntactic approach. In *Algebraic and Logic Programming, 6th International Joint Conference, ALP '97 - HOA '97, Southampton, UK, Spetember 3-5, 1997, Proceedings*, pages 174–193, 1997.

[5] Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

[6] Frédéric Blanqui. Inductive types in the calculus of algebraic constructions. *Fundamenta Informaticae*, 65(1-2):61–86, 2005.

[7] Frédéric Blanqui. Size-based termination of higher-order rewriting. *J. Funct. Program.*, 28:e11, 2018.

[8] Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, pages 9:1–9:21, 2019.

[9] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1):41–68, 2002.

[10] Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic strong normalization and confluence. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca, editors, *ICALP 1989: Automata, Languages and Programming*, pages 137–150, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[11] Carsten Fuhs and Cynthia Kop. Polynomial interpretations for higher-order rewriting. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, pages 176–192, 2012.

[12] Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[13] Cynthia Kop and Femke van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Log. Methods Comput. Sci.*, 8(2), 2012.

[14] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159–172, 1991.

[15] Mitsuhiro Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, ISSAC '89, Portland, Oregon, USA, July 17-19, 1989*, pages 357–363, 1989.

[16] Erik Palmgren. On universes in type theory. In Giovanni Sambin and Jan M. Smith, editors, *Twenty Five Years of Constructive Type Theory*, Oxford Logic Guides, pages 191–204. Oxford University Press, 1998.

[17] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In A. D. Gordon, editor, *Foundations of Software Science and Computation Structures. FoSSaCS 2003*, pages 425–440, 2003.

[18] Daria Walukiewicz-Chrząszcz. Termination of rewriting in the calculus of constructions. *J. Funct. Program.*, 13(2):339–414, 2003.

Semantic Analysis of Normalization by Evaluation for Fitch-Style Modal Lambda Calculi

Nachiappan Valliappan[1], Fabian Ruch, and Carlos Tomé Cortiñas[1]

[1]Chalmers University of Technology

Fitch-style modal lambda calculi (Borghuis 1994; Clouston 2018) provide a solution to programming necessity modalities (denoted by a $\Box$) in a typed lambda calculus by extending the typing context with a delimiting operator (denoted by a 🔒). In this work, we perform a semantic analysis of *normalization by evaluation* (NbE) (Berger and Schwichtenberg 1991) for Fitch-style modal lambda calculi by beginning with the calculus λ_{IK}—a system for the most basic modal logic IK (for "intuitionistic" and "Kripke")—as our object of study. We construct an NbE model for λ_{IK}, and show that it is an instance of the possible-worlds semantics for IK. The presented NbE procedure has been formalized (Valliappan 2020–2021) in the proof assistant Agda (Abel et al. 2005–2021).

The Fitch-style modal lambda calculus under consideration. IK extends intuitionistic propositional logic with the *necessity modality* $\Box$, the *necessitation rule* (if $\cdot \vdash A$ then $\Gamma \vdash \Box A$) and the *K axiom* ($\Box(A \to B) \to \Box A \to \Box B$). Correspondingly, λ_{IK} extends the simply-typed lambda calculus (STLC) with the typing rules in Figure 1. The rules for λ-abstraction and function application are formulated in the usual way—but note the modified variable rule!

$$\text{Ty} \quad A ::= \ldots \mid \Box A \qquad\qquad \text{Ctx} \quad \Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, 🔒$$

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ 🔒 \notin \Gamma' \qquad \frac{\Gamma, 🔒 \vdash t : A}{\Gamma \vdash \mathrm{box}\ t : \Box A} \qquad \frac{\Gamma \vdash t : \Box A}{\Gamma, 🔒, \Gamma' \vdash \mathrm{unbox}\ t : A}\ 🔒 \notin \Gamma'$$

Figure 1: Typing rules for λ_{IK} (omitting λ-abstraction and application)

The NbE model for λ_{IK}. NbE is the process of *evaluating*, or interpreting, terms of a calculus in a suitable model and then *reifying*, or extracting, normal forms from values in that model. NbE for STLC can be performed by interpreting types and contexts as *covariant* presheaves over the category W of contexts Γ, Δ and order-preserving embeddings (OPEs) $e : \Gamma \leq \Delta$, and terms as natural transformations (Altenkirch, Hofmann, and Streicher 1995). Given that the category of presheaves $\widehat{W}$ is a cartesian closed category (CCC), the evaluation function $(\!|\_|\!) : \Gamma \vdash A \to [\![\Gamma]\!] \dot{\to} [\![A]\!]$ is given by the standard interpretation of STLC in a CCC. The reification function, on the other hand, is given by a family of natural transformations $\downarrow^A : [\![A]\!] \dot{\to} \mathrm{Nf}\ A$, where the presheaf $\mathrm{Nf}\ A$ denotes normal forms of type A.

To achieve NbE for λ_{IK}, we define a new category $W_🔒$ akin to W by requiring that morphisms additionally preserve locks and refer to the resulting notion of context embedding as *OLPE*. Note that whenever there is an OLPE $e : \Gamma \leq \Delta$ then Δ has the exact same number of locks as Γ. Further, we extend the interpretation of types and contexts to the type former $\Box$ and the context operator 🔒. Clouston (2018) shows that λ_{IK} can be soundly interpreted in a CCC equipped with an adjunction Lock $\dashv$ Box of endofunctors by interpreting $\Box$ by the right adjoint Box and 🔒 by the left adjoint Lock. Following this soundness result, we can use the CCC $\widehat{W_🔒}$ as our new NbE model, after equipping it with an adjunction. By virtue of

our definition of this adjunction (given in Figure 2), the evaluation of box and unbox is given by the *generic* interpreter of Clouston (2018), and we can construct natural transformations $\downarrow^{\Box A} : \mathrm{Box}\ [\![A]\!] \dot{\to} \mathrm{Nf}\ \Box A$, for every type A—thus retaining reification.

We summarize the data part of the NbE model for the modal fragment of λ_{IK} in Figure 2 as definitions in a constructive type-theoretic metalanguage. A presheaf $\mathcal{A}$ over $W_{\text{🔒}}$ consists of a family of sets $\mathcal{A}_\Gamma$ indexed by contexts Γ, and a family of functions $\mathrm{wk}_e : \mathcal{A}_\Gamma \to \mathcal{A}_{\Gamma'}$ indexed by OLPEs $e : \Gamma \leq \Gamma'$. The *reflection* function $\uparrow^A$ defines a natural transformation from the presheaf of neutral terms $\mathrm{Ne}\, A$, and can be used to construct an element $\mathrm{id}^\Gamma_{\mathrm{s}} : [\![\Gamma]\!]_\Gamma$. Normalization for a term $\Gamma \vdash t : A$ is then given by $\downarrow^A_\Gamma (([\![\mathrm{t}]\!])(\mathrm{id}^\Gamma_{\mathrm{s}}))$.

$$\frac{x : \mathcal{A}_{\Gamma,\text{🔒}}}{\mathbf{box}\ x : \mathrm{Box}_\Gamma\, \mathcal{A}} \qquad \frac{x : \mathcal{A}_\Gamma}{\mathbf{lock}\ x : \mathrm{Lock}_{\Gamma,\text{🔒},\Gamma'}\, \mathcal{A}}\ \text{🔒} \notin \Gamma'$$

$$[\![\_]\!] : \mathrm{Ty} \to \widehat{W_{\text{🔒}}}$$
$$[\![\Box A]\!]_\Gamma = \mathrm{Box}_\Gamma\ [\![A]\!]$$
$$[\![\_]\!] : \mathrm{Ctx} \to \widehat{W_{\text{🔒}}}$$
$$[\![\Delta, \text{🔒}]\!]_\Gamma = \mathrm{Lock}_\Gamma\ [\![\Delta]\!]$$

$$\begin{aligned}
(\!(_)\!) &: \Gamma \vdash A \to [\![\Gamma]\!]_\Delta \to [\![A]\!]_\Delta \\
(\!(\mathrm{box\ t})\!)\ \gamma &= \mathbf{box}\ ((\!(\mathrm{t})\!)\ \gamma) \\
(\!(\mathrm{unbox\ t})\!)\ \langle \gamma, _ \rangle &= (\!(\mathrm{unbox\ t})\!)\ \gamma \\
(\!(\mathrm{unbox\ t})\!)\ (\mathbf{lock}\ \gamma) &= \mathrm{wk}\, x \\
\quad \text{where } \mathbf{box}\ x &= (\!(\mathrm{t})\!)\ \gamma
\end{aligned}$$

$$\begin{aligned}
&\downarrow^A_\Gamma : [\![A]\!]_\Gamma \to \mathrm{Nf}_\Gamma\, A \\
&\downarrow^{\Box A}_\Gamma (\mathbf{box}\ x) = \mathrm{box}\ (\downarrow^A_{\Gamma,\text{🔒}} x) \\
&\uparrow^A_\Gamma : \mathrm{Ne}_\Gamma\, A \to [\![A]\!]_\Gamma \\
&\uparrow^{\Box A}_\Gamma n = \mathbf{box}\ (\uparrow^A_{\Gamma,\text{🔒}} (\mathrm{unbox}\, n))
\end{aligned}$$

Figure 2: NbE for the modal fragment of λ_{IK}

Connection with possible-worlds semantics. Analogously to how the NbE model for STLC can be seen as an instance of the Kripke semantics of IPL, the NbE model we present here can be seen as an instance of the possible-worlds semantics of IK. Hence, the observation that the NbE model construction for STLC corresponds to the completeness proof for Kripke semantics (C. Coquand 1993; T. Coquand and Dybjer 1997) carries over to the setting here.

The possible-worlds semantics for IK is parameterized by a *frame*, i.e. a type W together with two binary relations $\leq$ and R on W which are required to satisfy certain conditions (Božić and Došen 1984; Došen 1985; Simpson 1994): 1. $\leq$ is reflexive, 2. $\leq$ is transitive, 3. if $w \leq w'$ and $w'\ R\ v'$ then there exists $v : W$ such that $w\ R\ v$ and $v \leq v'$, and 4. if $w\ R\ v$ and $v \leq v'$ then there exists $w' : W$ such that $w \leq w'$ and $w'\ R\ v'$. An element $w : W$ can be thought of as a representation of the "knowledge state" about some "possible world" at a certain point in time; $w \leq w'$ as representing an increase in knowledge; and $w\ R\ v$ as specifying accessibility of worlds from one another.

Given a frame $(W, \leq, R)$, the possible-worlds semantics interprets a formula A at $w : W$ as the presheaf $\mathcal{A}(w)$ over $(W, \leq)$. The interpretation of $\Box A$ at w is the type of functions p assigning an element $p(v) : \mathcal{A}(v)$ to every $v : W$ such that $w\ R\ v$. Note that, by virtue of the frame conditions, the interpretation of $\Box$ extends to a functor Box on the category of presheaves and that Box has a left adjoint Lock. Hence, the possible-worlds semantics fits into the semantic framework of Clouston (2018). The left adjoint Lock can be described directly as mapping $\mathcal{A}$ and $w : W$ to the type of pairs $\langle v, a \rangle$ where $v : W$ such that $v\ R\ w$ and $a : \mathcal{A}(v)$

Now, we observe that the NbE model for λ_{IK} can be seen as the possible-worlds model where we pick Fitch-style contexts for W, OLPEs for $\leq$, extensions by a 🔒 for R, i.e. $\Gamma\ R\ \Delta$ if and only if there exists Γ' such that $\text{🔒} \notin \Gamma'$ and $\Delta = \Gamma, \text{🔒}, \Gamma'$ (cf. Figures 1 and 2), and normal forms as the interpretation of base types. Note that the required frame conditions are satisfied.

References

Abel, Andreas (2019). *Normalization by Evaluation for Call-by-Push-Value — Towards a Modal-Logical Reconstruction of NbE*. URL: https://www.cse.chalmers.se/~abela/talkTYPES2019.pdf.

Abel, Andreas et al., *Agda 2* version 2.6.1.3, 2005–2021. Chalmers University of Technology and Gothenburg University. LIC: BSD3. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php, VCS: https://github.com/agda/agda.

Alechina, Natasha et al. (2001). "Categorical and Kripke Semantics for Constructive S4 Modal Logic". In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, pp. 292–307. DOI: 10.1007/3-540-44802-0_21. URL: https://doi.org/10.1007/3-540-44802-0%5C_21.

Altenkirch, Thorsten, Martin Hofmann, and Thomas Streicher (1995). "Categorical Reconstruction of a Reduction Free Normalization Proof". In: *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*. Ed. by David H. Pitt, David E. Rydeheard, and Peter T. Johnstone. Vol. 953. Lecture Notes in Computer Science. Springer, pp. 182–199. DOI: 10.1007/3-540-60164-3_27. URL: https://doi.org/10.1007/3-540-60164-3%5C_27.

Bak, Miëtek (2017). *An Agda formalisation of a normalisation by evaluation procedure for the $\lambda_{\Box}$-calculus*. URL: https://github.com/mietek/imla2017.

Berger, Ulrich and Helmut Schwichtenberg (1991). "An Inverse of the Evaluation Functional for Typed lambda-calculus". In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, pp. 203–211. DOI: 10.1109/LICS.1991.151645. URL: https://doi.org/10.1109/LICS.1991.151645.

Boespflug, Mathieu and Brigitte Pientka (2011). "Multi-level Contextual Type Theory". In: *Proceedings Sixth International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2011, Nijmegen, The Netherlands, August 26, 2011*. Ed. by Herman Geuvers and Gopalan Nadathur. Vol. 71. EPTCS, pp. 29–43. DOI: 10.4204/EPTCS.71.3. URL: https://doi.org/10.4204/EPTCS.71.3.

Borghuis, V.A.J. (1994). "Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus". PhD thesis. Mathematics and Computer Science. DOI: 10.6100/IR427575.

Božić, Milan and Kosta Došen (1984). "Models for normal intuitionistic modal logics". In: *Studia Logica* 43.3, pp. 217–245. ISSN: 0039-3215. DOI: 10.1007/BF02429840. URL: https://doi.org/10.1007/BF02429840.

Clouston, Ranald (2018). "Fitch-Style Modal Lambda Calculi". In: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Lecture Notes in Computer Science. Springer, pp. 258–275. DOI: 10.1007/978-3-319-89366-2_14. URL: https://doi.org/10.1007/978-3-319-89366-2%5C_14.

Coquand, Catarina (1993). "From Semantics to Rules: A Machine Assisted Analysis". In: *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*. Ed. by Egon Börger, Yuri Gurevich, and Karl Meinke. Vol. 832. Lec-

ture Notes in Computer Science. Springer, pp. 91–105. DOI: 10.1007/BFb0049326. URL: https://doi.org/10.1007/BFb0049326.

Coquand, Thierry and Peter Dybjer (1997). "Intuitionistic Model Constructions and Normalization Proofs". In: *Math. Struct. Comput. Sci.* 7.1, pp. 75–94. DOI: 10.1017/S0960129596002150. URL: https://doi.org/10.1017/S0960129596002150.

Došen, Kosta (1985). "Models for stronger normal intuitionistic modal logics". In: *Stud Logica* 44.1, pp. 39–70. DOI: 10.1007/BF00370809. URL: https://doi.org/10.1007/BF00370809.

Gratzer, Daniel, Jonathan Sterling, and Lars Birkedal (2019). "Implementing a modal dependent type theory". In: *Proc. ACM Program. Lang.* 3.ICFP, 107:1–107:29. DOI: 10.1145/3341711. URL: https://doi.org/10.1145/3341711.

Plotkin, Gordon D. and Colin Stirling (1986). "A Framework for Intuitionistic Modal Logics". In: *Proceedings of the 1st Conference on Theoretical Aspects of Reasoning about Knowledge, Monterey, CA, USA, March 1986*. Ed. by Joseph Y. Halpern. Morgan Kaufmann, pp. 399–406.

Simpson, Alex K. (1994). "The proof theory and semantics of intuitionistic modal logic". PhD thesis. University of Edinburgh, UK. URL: http://hdl.handle.net/1842/407.

Valliappan, Nachiappan (2020–2021). *Mechanisation of Fitch-style Intuitionistic K in Agda*. URL: https://github.com/nachivpn/k.

Type-Theoretic Constructions of the Final Coalgebra of the Finite Powerset Functor

Niccolò Veltri

Tallinn University of Technology, Estonia
niccolo@cs.ioc.ee

The powerset functor, delivering the set of subsets of a given set, plays a fundamental role in the behavioral analysis of nondeterministic systems [9], which include process calculi such as Milner's calculus of communicating systems and π-calculus. A nondeterministic system is determined by a function $c : S \to \mathsf{P}\, S$, called a coalgebra, from a set of states S to the set $\mathsf{P}\, S$ of subsets of S. The function c associates to each state $x : S$ a set of new states $c\, x$ reachable from x, so it represents the transition relation of an unlabelled transition system. Adding labels to transitions is easy, just consider coalgebras of the form $c : S \to \mathsf{P}\, (A \times S)$ or $c : S \to (A \to \mathsf{P}\, S)$ instead, where A is a set of labels. In many applications, the set of reachable states is known to be finite, so the powerset functor P can be replaced by the finite powerset functor Pfin delivering only the set of finite subsets.

The behavior of a finitely nondeterministic system starting from a given initial state is fully captured by the final coalgebra of Pfin. Elements of the final coalgebra are execution traces obtained by iteratively running the coalgebra function modelling the system on the initial state. The resulting traces are possibly infinite trees with finite unordered branching. Several formal constructions of the final coalgebra of Pfin and other finitary set functors exist in the literature, developed using various different techniques [5, 1, 12, 2]. Adámek et al. collect and compare all these characterizations in their recent book draft [3, Chapter 4]. All these constructions take place in set theory, and reasoning is based on classical logic.

In this work we present various definitions of the final coalgebra of the finite powerset functor in constructive type theory, which have all been formalized in the Cubical Agda proof assistant [11]. Cubical Agda is an implementation of cubical type theory [6], which in turns is a particular presentation of homotopy type theory with support for univalence and higher inductive types (HITs). The choice of Cubical Agda as our foundational setting, over other proof assistants based on Martin-Löf type theory or the calculus of constructions such as plain Agda, Coq or Lean, lies in the fact that both univalence and HITs play an important role for both encoding and reasoning with the finite powerset datatype in homotopy type theory [7]. In our development we also take advantage of Cubical Agda's support for coinductive types.

The formalization is available at https://github.com/niccoloveltri/final-pfin. More details can be found in an upcoming paper [10].

The final coalgebra using setoids. Given a setoid (A, R), its setoid of finite subsets is defined as $\mathsf{Pfin_s}\, (A, R) =_{\mathrm{df}} (\mathsf{List}\, A, \overline{\mathsf{List}}\ R)$, where $\overline{\mathsf{List}}$ is a lifting of List to relations. Given a type family $R : A \to A \to \mathsf{Type}$, the type family $\overline{\mathsf{List}}\ R : \mathsf{List}\, A \to \mathsf{List}\, A \to \mathsf{Type}$ is defined as

$$\begin{aligned} \overline{\mathsf{List}}\ R\, s\, t =_{\mathrm{df}} & ((x : A) \to x \in s \to \exists y : A.\, y \in t \times R\, x\, y) \\ & \times \\ & ((y : A) \to y \in t \to \exists x : A.\, x \in s \times R\, y\, x) \end{aligned}$$

So two lists are related by $\overline{\mathsf{List}}\ R$ when each element of a list is R-related to at least one element of the other list. Notice that $\overline{\mathsf{List}}$ is not the standard relation lifting on lists, it is often called a *relator*, and plays an important role in the study of applicative bisimilarity for

functional programming languages with nondeterministic choice [8]. The final coalgebra of Pfin_s in the category of setoids is the setoid composed of the final coalgebra of the list functor, whose elements are non-wellfounded finitely-branching trees, and the coinductive relation TreeR relating two trees if, for each subtree of one tree, there merely exists a TreeR-related subtree of the other tree. In Agda the definitions look as follows:

```
record Tree : Type where
  coinductive
  field
    subtrees_L : List Tree
```

```
record TreeR (t u : Tree) : Type where
  coinductive
  field
    subtreesR : List̄ TreeR (subtrees_L t) (subtrees_L u)
```
(1)

The final coalgebra using quotient inductive types. Given a type A, the type of its finite subsets $\mathsf{Pfin}\,A$ can be defined as the set quotient of $\mathsf{List}\,A$ by the relation $\overline{\mathsf{List}}\,(=)$. Equivalently, it can also be defined as a quotient inductive type, as the free join semilattice on A [7]. In Cubical Agda, the final coalgebra of Pfin can be given as the coinductive type on the left:

```
record νPfin : Type where
  coinductive
  field
    subtrees_P : Pfin νPfin
```

```
record νPfinB (t u : νPfin) : Type where
  coinductive
  field
    subtreesB_P : Pfin̄ νPfinB (subtrees_P t) (subtrees_P u)
```

The coinductive relation on the right is the notion of bisimilarity for $\nu\mathsf{Pfin}$, which crucially in Cubical Agda can be proved equivalent to path equality. The relation lifting $\overline{\mathsf{Pfin}}$ is defined analogously to $\overline{\mathsf{List}}$ in (1), where now s, t are finite subsets and the list membership relation $\in$ is replaced by the appropriate membership relation for finite subsets.

Alternatively, one can think of obtaining the final Pfin-coalgebra from the final Pfin_s-coalgebra in setoids, i.e. as the set quotient of the type Tree of coinductive trees by the equivalence relation TreeR. The resulting type $\mathsf{Tree/TreeR}$ is indeed a fixpoint of Pfin, i.e. $\mathsf{Pfin}\,(\mathsf{Tree/TreeR}) \simeq \mathsf{Tree/TreeR}$, but proving the finality of the coalgebra underlying this equivalence seems to require the assumption of the full axiom of choice.

Analysis of Worrell's classical set theoretic construction. It is well known that the chain of iterated applications of Pfin on the singleton set does not stabilize after ω steps [1]. This is in antithesis with the case of polynomial functors, whose final coalgebras (a.k.a. M-types in type theory) always arise as ω-limits, a fact that can also be proved in homotopy type theory [4]. James Worrell showed in classical set theory that the final Pfin-coalgebra can be obtained by iterating applications of Pfin for extra ω steps, i.e. as the $(\omega + \omega)$-limit of the chain [12]. Elements of the ω-limit are represented by non-wellfounded trees with unordered but possibly infinite branching, while the $(\omega + \omega)$-limit corresponds to the subset of these trees with finite branching at all levels.

In our constructive setting, Worrell's construction of the $(\omega+\omega)$-limit is indeed the final Pfin-coalgebra, modulo the assumption of classical principles such as the axiom of countable choice and the lesser limited principle of omniscience (LLPO). Notably, Worrell's iterated construction is inherently classical: the ω-limit is equipped with a canonical Pfin-algebra structure, but the injectivity of the latter is equivalent to LLPO. Concretely this means that it is impossible to prove that the $(\omega + \omega)$-limit is a subset of the ω-limit, as in Worrell's construction, without the assumption of LLPO.

Acknowledgments This work was supported by the Estonian Research Council grant PSG659 and by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001).

References

[1] Jiří Adámek and Václav Koubek. On the greatest fixed point of a set functor. *Theoretical Computer Science*, 150(1):57–75, 1995.

[2] Jiří Adámek, Paul Blain Levy, Stefan Milius, Lawrence S. Moss, and Lurdes Sousa. On final coalgebras of power-set functors and saturated trees. *Applied Categorical Structures*, 23(4):609–641, 2015.

[3] Jiří Adámek, Stefan Milius, and Lawrence S. Moss. Initial algebras, terminal coalgebras, and the theory of fixed points of functors. Draft book, available from http://www.stefan-milius.eu, 2021.

[4] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In Thorsten Altenkirch, editor, *Proc. of 13th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'15*, volume 38 of *Leibniz International Proceedings in Informatics*, pages 17–30. Schloss Dagstuhl, 2015.

[5] Michael Barr. Terminal coalgebras in well-founded set theory. *Theoretical Computer Science*, 114(2):299–315, 1993.

[6] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *Proc. of 21st Int. Conf. on Types for Proofs and Programs, TYPES'15*, volume 69 of *Leibniz International Proceedings in Informatics*, pages 5:1–5:34. Schloss Dagstuhl, 2015.

[7] Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. Finite sets in homotopy type theory. In *Proc. of 7th ACM SIGPLAN Int. Conf. on Certified Programs and Proofs, CPP'18*, pages 201–214. ACM, 2018.

[8] Paul Blain Levy. Similarity quotients as final coalgebras. In Martin Hofmann, editor, *Proc. of 14th Int. Conf on Foundations of Software Science and Computational Structures, FoSSaCS'11*, volume 6604 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2011.

[9] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[10] Niccolò Veltri. Type-theoretic constructions of the final coalgebra of the finite powerset functor. In Naoki Kobayashi, editor, *Proc. of 6th Int. Conf. on Formal Structures for Computation and Deduction, FSCD'21*, to appear.

[11] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *PACMPL*, 3(ICFP):87:1–87:29, 2019.

[12] James Worrell. On the final sequence of a finitary set functor. *Theoretical Computer Science*, 338(1-3):184–199, 2005.

A drag-and-drop proof tactic

Pablo Donato Pierre-Yves Strub Benjamin Werner
LIX, Ecole polytechnique, France

June 2021

Introduction

Interactive Theorem Provers allow the user to incrementally construct formal proofs through an interaction loop. One progresses through a sequence of *states* corresponding to incomplete proofs. Each of these states is itself described by a finite set of *goals* and the proof is completed once there are no goals left. From the user's point of view, a goal appears as a sequent, in the sense coined by Gentzen. In the case of intuitionistic logic that is:

- One particular proposition A which is *to be proved*, we designate it as the goal's *conclusion*,
- a set of propositions Γ corresponding to *hypotheses*.

The user performs *actions* on one such goal at a time, and the actions transform the goal, or rather replace the goal by a new set of goals. When this set is empty, the goal is said to be solved.

In the dominant paradigm, these commands are provided by the user in text form; since Robin Milner and LCF [3] they are called *tactics*.

The present work is a form of continuation of the *Proof-by-Pointing* (PbP) effort, initiated in the 1990ties by Gilles Kahn, Yves Bertot, Laurent Théry and their group [1]. Both works share a main idea which is to replace the textual tactic commands by *physical actions* performed by the user on a graphical user interface. In both cases, the *items* the user performs actions on are the current goal's conclusion and hypotheses. What is new in our work is that we allow not only to *click* on subterms of these items, but also to *drag-and-drop* (DnD) one subterm onto another. This enriches the language of actions in, we argue, an intuitive way. We should point out that what is proposed here is not meant to replace but to complement the proof-by-pointing features. We thus envision a general *proof-by-action* paradigm, which includes both PbP and DnD features.

We have implemented a small web-based prototype to demonstrate and explore this approach. We think there is a possibility this will help in making proof systems more accessible and user-friendly, among other fields in education.

Setting

The proof-by-action approach ought to be applied to various formalisms; the current prototype implements first-order intuitionistic logic. One advantage of this approach is that it allows a very lean visual layout of the proof state; a goal thus appears as a set of items, whose nature is defined by their respective colors:

- A *red item* which is the proposition to be proved, that is the *conclusion*,
- *blue items*, which are the local *hypotheses*.

Each goal is displayed on a tab. In these tabs, each item thus appears as a red or blue rectangle, bearing a logical *statement* of the item. By default, the red conclusion statement is on the right and the blue

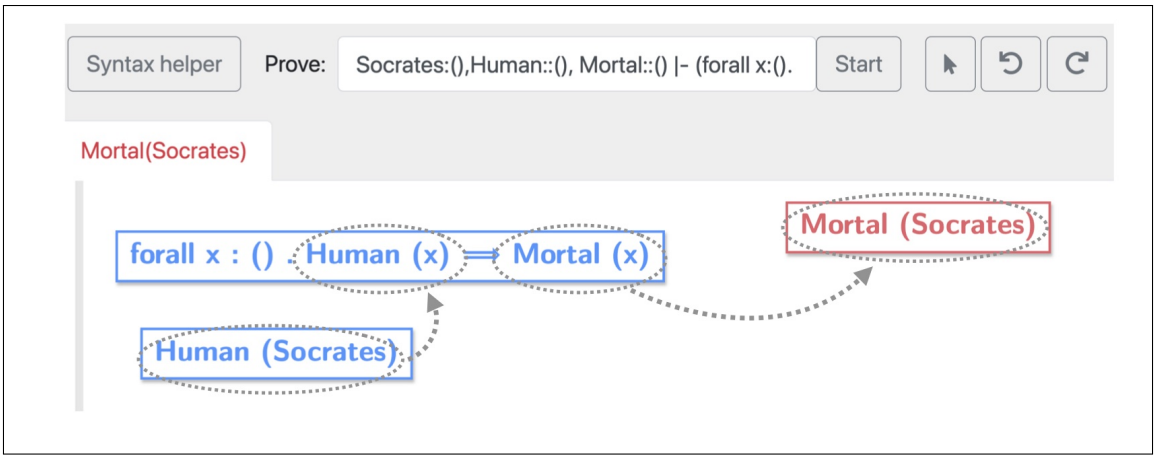


Figure 1: A partial screenshot showing a goal in the Actema prototype. The conclusion is red on the right, the two hypotheses blue on the left. The gray dotted lines and arrows have been added to show two possible drag-and-drop actions.

hypotheses are on the left; see figure 1 for a possible state.

The items are what the user can act upon: either by clicking *on them, or by* moving *them.*

The intuition behind the drag-and-drop proof tactic is, we hope, simple. It builds on the distinction between the roles red and blue items play in the proof:

- the red conclusion A *demands* evidence that the proposition A is true;
- on the other hand, a blue hypothesis B *provides* evidence that B is true in the considered state.

This materializes in the most basic DnD proof construction action. Given a goal whose conclusion is a proposition A, if this goal also yields a hypothesis A, then one can *drag* one A and drop it onto the other, which solves the goal.

We hope that the intuition of this proof construction step is reasonably clear: grab the evidence (the hypothesis) and bring it where it is needed (the conclusion); or conversely, grab the proposition to be proved, and bring it to some evidence. Much of the work is about how to generalize this idea to more complex situations while trying to stick to an intuitive behavior. From a proof-theoretical point of view, this can thus be understood as an attempt to provide a generalized axiom rule and appears related to deep inference [2]; understanding this helped to design the precise behavior of the tactic.

Note that some actions will not solve the goal but transform the conclusion while others will combine two hypotheses in order to create a new one. In the case illustrated in figure 1 for instance, there are two possible actions:

- One can drop the upper hypothesis on the conclusion; this transforms the conclusion to Human(Socrates) (and the goal can then be solved by using the other hypothesis).
- If one drops the hypothesis Human(Socrates) on the other one, this gives birth to a new fact/hypothesis Mortal(Socrates).

References

[1] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789, pages 141–160. Springer Berlin Heidelberg, 1994. Series Title: Lecture Notes in Computer Science.

[2] Kaustuv Chaudhuri. Subformula linking as an interaction method. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998, pages 386–401. Springer Berlin Heidelberg, 2013. Series Title: Lecture Notes in Computer Science.

[3] Robin Milner. The use of machines to assist in rigorous proof. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 312(1522):411–422, 1984.

Native Type Theory (Extended Abstract)

Christian Williams[1] and Michael Stay[2]

[1] UC Riverside, California, US
cwill041@ucr.edu
[2] Pyrofex Corp., Linden, Utah
stay@pyrofex.net

Abstract

We present a method to construct "native" type systems for a broad class of languages, in which types are built from term constructors by predicate logic and dependent types. Many languages can be modelled as structured λ-theories, and the internal language of their presheaf toposes provides total specification the structure and behavior of programs. The construction is functorial, thereby providing a shared framework of higher-order reasoning for most existing programming languages. The full paper is on arXiv [6].

Introduction

Type theory is growing as a guiding philosophy in the design of programming languages, but in practice type systems are mostly heterogeneous, and there are not standard ways to reason across languages. We construct from a typed λ-calculus a *native type system* which provides total specification of the structure and behavior of terms.

Categorical logic constructions can be composed to generate expressive type systems:

$$\lambda\mathtt{theory} \xrightarrow{\mathcal{P}} \mathtt{topos} \xrightarrow{\mathcal{L}} \mathtt{type\ system}$$

The first is the *presheaf construction* $\mathcal{P}$ [2, Ch. 8]; it preserves product, equality, and function types. The second is the *language of a topos* $\mathcal{L}$ [3, Ch. 11]. The composite is 2-functorial, so that translations between languages induce translations between type systems.

We aim to implement native type theory as a tool which inputs the formal semantics of a language and outputs a type system, which can then be used to condition codebases. The semantics of languages such as JavaScript, C, Java, Python, Haskell, LLVM and Solidity [1] can be represented as λ-theories with rewrites. Properly integrated, native type systems could introduce expressive type theory directly into everyday programming.

λ-Theories

Typed λ-calculus is the internal language of cartesian closed categories [4]; these model languages with product and function types. A useful addition is *equality types*: we define a **λ-theory with equality** to be a cartesian closed category with pullbacks. Our leading example is a concurrent language called the ρ-calculus, or **r**eflective **h**igher-**o**rder π-calculus [5].

$$\begin{array}{llll} \mathtt{0}: & 1 \to \mathtt{P} & -|-: & \mathtt{P} \times \mathtt{P} \to \mathtt{P} \\ @: & \mathtt{P} \to \mathtt{N} & \mathtt{out}: & \mathtt{N} \times \mathtt{P} \to \mathtt{P} \\ *: & \mathtt{N} \to \mathtt{P} & \mathtt{in}: & \mathtt{N} \times [\mathtt{N},\mathtt{P}] \to \mathtt{P} \end{array}$$

$$s,t: \ \mathtt{E} \to \mathtt{P} \qquad \mathtt{comm}: \ \mathtt{N} \times \mathtt{P} \times [\mathtt{N},\mathtt{P}] \to \mathtt{E}$$

$$\begin{array}{lcl} s(\mathtt{comm}(n,q,\lambda x.p)) & = & \mathtt{out}(n,q)|\mathtt{in}(n,\lambda x.p) \\ t(\mathtt{comm}(n,q,\lambda x.p)) & = & p[@q/x] \end{array}$$

The Logic of a Presheaf Topos

A λ-theory embeds into a presheaf topos by the Yoneda embedding, and the internal language of the topos constitutes its native type system.

A presheaf is a context-indexed set of data on the sorts of a theory. The canonical example is a representable presheaf, of the form $\mathrm{T}(-, \mathtt{S})$, which indexes all terms of sort $\mathtt{S}$. A predicate $\varphi : \mathrm{T}(-, \mathtt{S}) \to \mathsf{Prop}$ is a shape of abstract syntax tree in T, or a type of program structure.

A *higher-order dependent type theory* is a pair of fibrations $p : Pred \to \mathcal{E}$ and $q : Type \to \mathcal{E}$ connected by a fibered reflection $i \dashv c : Pred \leftrightharpoons Type$ [3, Ch.11]. These form a 2-category HDTΣ, with morphisms of fibered adjunctions.

Theorem. There is a 2-functor $\mathcal{L} : \mathrm{Topos} \to \mathrm{HDT}\Sigma$ which sends $\mathcal{E}$ to its predicate and codomain fibrations, connected by image and comprehension. This sends a topos to its internal language, given by predicate logic and dependent type theory.

All together, **native type theory** is a 2-functor

$$\lambda\mathrm{Thy}_{=}^{\mathrm{op}} \xrightarrow{\mathcal{P}} \mathrm{Topos} \xrightarrow{\mathcal{L}} \mathrm{HDT}\Sigma.$$

The construction freely generates a highly expressive type system for the language of T. The types are *native* in that they are built from term constructors and dependent type constructors. This enables higher-order reasoning which is intrinsic to the language.

Native Type Theory

The **native type system** of a theory T is $\mathcal{L}\mathcal{P}(T)$, the higher-order dependent type system of $\mathcal{P}(\mathrm{T})$. Types $\mathsf{x}{:}\mathsf{A} \vdash \mathsf{B}(\mathsf{x}) : \mathsf{Type}$ are indexed presheaves $f : B \to A$.

The basic types are $\mathrm{T}(-, \mathtt{T}) \vdash \mathrm{T}(-, \mathtt{f}) : \mathsf{Type}$ for each operation $f : \mathtt{S} \to \mathtt{T}$. The main type constructors are dependent sum and dependent product, which generalize quantification and enable higher-order reasoning in the language of T.

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}.\mathsf{B} : \mathsf{Type}}\ \Sigma_{\mathrm{F}} \qquad \frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{B} : \mathsf{Type}}\ \Pi_{\mathrm{F}}$$

The system has inductive and coinductive types, equality and quotient types, and more. Most importantly, by adding rules for functoriality, we can reason across translations.

Applications

As a small example of the expressiveness of native types, let T be the theory of the ρ-calculus. The graph of rewrites over processes is a type $\mathrm{T}(-, \mathtt{P}^2) \vdash \mathrm{g} := \mathrm{T}(-, \langle s, t \rangle) : \mathsf{Type}$.

$$\mathrm{g}(\mathtt{S})(p_1, p_2) = \{e \mid \mathtt{S} \vdash e : p_1 \rightsquigarrow p_2\}$$

This type is the space of all computations in the ρ-calculus. The native type system can filter to subspaces: below is the type of communications occuring on names in α, sending data in φ, and for $F : [\mathtt{N}, \mathsf{Prop}] \to [\mathtt{P}, \mathsf{Prop}]$ continuing in contexts $\lambda x.c : [\mathtt{N}, \mathtt{P}]$ such that $\chi(n) \Rightarrow F(\chi)(c[n/x])$.

$$\Sigma e{:}\mathtt{comm}(\alpha, \varphi, \chi.F).\mathrm{g}$$

We can then construct modalities relative to these subspaces, as well as behavioral equivalences. By including a graph of rewrites, native type theory can reason about not only the structure but also the behavior of terms, and explore their interconnection.

References

[1] K framework. http://www.kframework.org/.

[2] S. Awodey. *Category Theory*. Oxford University Press, Inc., USA, 2nd edition, 2010.

[3] B. Jacobs. *Categorical Logic and Type Theory*. Elsevier, Amsterdam, 1998.

[4] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, USA, 1986.

[5] L.G. Meredith and Matthias Radestock. A reflective higher-order calculus. *Electronic Notes in Theoretical Computer Science*, 141(5):49–67, dec 2005.

[6] C. Williams and M. Stay. Native type theory. https://arxiv.org/abs/2102.04672, 2021.

Shape-irrelevant reflection: Terminating extensional type theory.

Théo Winterhalter

MPI-SP, Bochum, Germany
theo.winterhalter@mpi-sp.org

Abstract

We introduce a variant of extensional type theory where *reflection* is restricted to shape-irrelevant subterms. This extends the notion of definitional irrelevance while retaining nice properties like strong normalisation and non-confusion. With this, we propose a solution to dealing with the infamous indices in dependent type theory, but also some natural meta-theory for the use of automation like SMT-solvers to produce (shape-)irrelevant proofs in the style of F*.

Extensional type theories are dependent type theories extended with the so-called *reflection* of equality rule. It states that whenever two terms u and v are *provably* (or propositionally) equal, then they are *convertible* (or defintionally equal). This can be very convenient when reasoning or programming because equalities do not 'get in the way'; the type system ensures that they must have been checked, but they do not appear explicitly in terms. As such, several systems like F* [10], Andromeda 1 [4] and NuPRL [6] implement variants of extensional type theories.

$$\frac{\Gamma \vdash p : u =_A v}{\Gamma \vdash u \equiv v : A}$$

These advantages come at a cost however: in presence of the reflection rule, type checking becomes undecidable. Despite its apparent convenience it is thus absent from proof assistants like Coq [11] and Agda [8] which prefer a more controlled notion of conversion and typing. Decidability is not the only aspect that is relevant here: another one is the notion of confusion, or rather lack thereof. In presence of the reflection rule, any two terms are convertible in an inconsistent context. For instance a product type can be confused with the type of natural numbers. We thus lose the good properties expected of a programming language as data is no longer necessarily of the expected form.

Irrelevance. From these observations, it seems natural to have reflection only happen in ways that do not affect the data representation. This would point to a use of reflection limited to computationally irrelevant arguments [9]. Abel and Scherer [1] already developed a type theory with irrelevant arguments that are ignored by conversion. The theory is extended with irrelevant binders (written $\div$ rather than the usual colon) and the fact that such arguments are ignored for conversion. More recently [7] have introduced a proof-irrelevant sort in Coq and Agda which contains propositions, the proofs of which are always convertible. Of course, reflection on such irrelevant terms does not really make sense because they are all already identified. Thus, we propose to have reflection for *shape-irrelevant* subterms.

Shape-irrelevance. An argument to a function is shape-irrelevant when it does not affect its shape. A prototypical example of a *non*-shape-irrelevant argument is a natural number used to produce an n-ary dependent product type, another one is the size argument in sized-types [2]. For our purposes, we will consider the natural number index n in the type of vectors $\mathsf{vec}_A\ n$, as well as the predicate P in subset type $\{x : A \mid P\}$ to be shape-irrelevant. $\mathsf{vec}_A\ n$ describes the type of lists of type A whose length is equal to n. An inhabitant of $\mathsf{vec}_A\ n$ and one of $\mathsf{vec}_A\ m$ are both represented as lists but might have different lengths. Identifying $\mathsf{vec}_A\ n$ and $\mathsf{vec}_A\ m$ would not make much sense (otherwise this type would merely be that of list_A). What we propose however is to consider the index up to reflection.

Shape-irrelevant reflection type theory (SIRTT) We define and formalise [13] in Coq a type theory we call SIRTT which features relevance annotations ($\mathfrak{r}$, $\mathfrak{s}$ or $\mathfrak{i}$) on binders, an equality type, a type of vectors, as well as refinement types. Relevance levels (written ℓ when abstract) $\mathfrak{r}$, $\mathfrak{s}$ and $\mathfrak{i}$ stand respectively for 'relevant', 'shape-irrelevant' and 'irrelevant', and also annotate typing and conversion with the idea that relevant data can be used in place of shape-irrelevant data, and shape-irrelevant data in place of

irrelevant data. Furthermore, when stating $\Gamma \vdash t :_\ell A$, while t lives in Γ as usual, the type A lives in a modified version of Γ where all irrelevant binders have been replaced by shape-irrelevant ones, written $\Gamma^{\blacktriangle}$. This property shares some similary with quantitative type theory [3]. The crucial point of having this annotation is to impact conversion. The most interesting conversion rules are below.

$$\frac{}{\Gamma \vdash u \equiv_{\mathfrak{i}} v} \qquad \frac{\Gamma \vdash p :_{\mathfrak{i}} u =_A v}{\Gamma \vdash u \equiv_{\mathfrak{s}} v} \qquad \frac{\Gamma \vdash A \equiv_{\mathfrak{r}} B \qquad \Gamma \vdash n \equiv_{\mathfrak{s}} m}{\Gamma \vdash \mathsf{vec}_A\ n \equiv_{\mathfrak{r}} \mathsf{vec}_B\ m}$$

$$\frac{\Gamma \vdash A \equiv_{\mathfrak{r}} B \qquad \Gamma, x :_{\mathfrak{r}} A \vdash P \equiv_{\mathfrak{s}} Q}{\Gamma \vdash \{x : A \mid P\} \equiv_{\mathfrak{r}} \{x : B \mid Q\}} \qquad \frac{\Gamma \vdash f \equiv_{\mathfrak{r}} g \qquad \Gamma \vdash u \equiv_\ell v}{\Gamma \vdash f\ @_\ell\ u \equiv_{\mathfrak{r}} g\ @_\ell\ v}$$

The first rule states that irrelevant subterms are always convertible, the second corresponds to shape-irrelevant reflection, while the others show how the different layers interact via congurence rules of vector types, refinement types and applications (at arbitrary relevance level ℓ) respectively.

For the language to be interesting we need not only a way to go from relevant to irrelevant but also a restricted arrow in the other direction. This is similar to how singleton elimination is necessary for the universe of proposition of Coq to be used in a relevant context. We bridge this gap with the empty type $\bot$ or more precisely with its eliminator exfalso which expects an *irrelevant* proof of $\bot$ to relevantly inhabit any type.

$$\frac{\Gamma^{\blacktriangle} \vdash A \qquad \Gamma \vdash p :_{\mathfrak{i}} \bot}{\Gamma \vdash \mathsf{exfalso}_A\ p :_{\mathfrak{r}} A}$$

This language features computation with for instance a β-reduction rule for relevant application, but crucially, it doesn't feature such *reduction*[1] for (shape-)irrelevant applications. To account for the fact that a redex might be *hidden* under irrelevant redexes, we introduce a function which consumes these redexes without applying the substitution (and thus is trivially terminating) which we write in a relational way as $t \triangleright u \mid \sigma$ to mean that t *reveals* relevant term u and irrelevant substitution σ. The idea is that we do not need to apply the substitution σ to compute since its content are anyway computationally irrelevant.

$$\frac{t \triangleright v \mid \sigma}{(\lambda(x :_{\mathfrak{i}} A).\ t)\ @_{\mathfrak{i}}\ u \triangleright v | \sigma, x \mapsto u} \qquad \frac{t \triangleright \lambda(x :_{\mathfrak{r}} A).\ b \mid \sigma}{t\ @_{\mathfrak{r}}\ u \rightsquigarrow b\sigma[x \mapsto u]}$$

Thanks to this, we are able to show that one-step reduction ($\rightsquigarrow$) is preserved by an erasure program transformation, which constitutes the main contribution.

Erasure of SIRTT to MLTT We formalise a notion of MLTT with lists, natural numbers and equality and an axiom for the empty type ($\bot$). Even if this theory is trivially inconsistent, it still enjoys properties like strong normalisation and non-confusion (which we do not formalise) which can be lifted back to SIRTT thanks to an erasure translation (which we will write with square brackets in the style of [5]).

The idea of erasure is to simply remove all (shape-)irrelevant subterms, for instance erasing vectors to lists: $[\mathsf{vec}_A\ n] := \mathsf{list}_{[A]}$. The need for axiom $\bot$ comes when erasing exfalso for which the proof is irrelevant and is thus erased and replaced by the use of an axiom. It is similar in a sense to how extraction/compilation would replace static checks by dynamic failures.

Theorem 1. *Erasure from SIRTT to MLTT preserves reduction, conversion and typing:*

1. *If* $u \rightsquigarrow v$ *then* $[u] \rightsquigarrow [v]$.
2. *If* $\Gamma \vdash u \equiv_{\mathfrak{r}} v$ *then* $[u] \equiv [v]$.
3. *If* $\Gamma \vdash t :_{\mathfrak{r}} A$ *then* $[\![\Gamma]\!] \vdash [t] : [\![A]\!]$.

Corollary 2. *Assuming MLTT enjoys strong normalisation and non-confusion, so does SIRTT.*

From this, we cannot obtain consistency of SIRTT. As future work, we plan to explore another translation from SIRTT to MLTT but without $\bot$, in the style of [12]. We believe this could take the form of a type-checker for SIRTT returning a set of equalities to be proven and a term in MLTT using transports.

[1] It still features a conversion rule for those.

References

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

[2] Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *Proceedings of the ACM on Programming Languages*, 1:33, 2017.

[3] Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 56–65, 2018.

[4] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Chris Stone. The 'Andromeda' prover, 2016.

[5] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Certified Programs and Proofs – CPP 2017*, pages 182–194, January 2017.

[6] Robert L. Constable and Joseph L. Bates. The NuPrl system, PRL project, 2014.

[7] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without k. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[8] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

[9] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 221–230. IEEE, 2001.

[10] Nikhil Swamy, Catalin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, and Jean-Karim Zinzindohoue. Dependent types and multi-monadic effects in F*. Draft, July 2015.

[11] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2020. Version 8.11.

[12] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. Eliminating Reflection from Type Theory. In *CPP 2019 - 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 91–103, Lisbonne, Portugal, January 2019. ACM.

[13] Théo Winterhalter. Formalisation of SIRTT, 2021. https://github.com/TheoWinterhalter/sirtt.