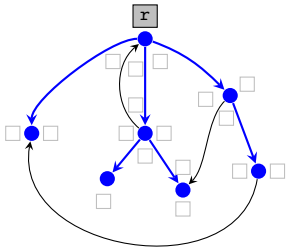
A Datatype of Planar Graphs

Malin Altenmüller and Conor McBride

University of Strathclyde, Glasgow, United Kingdom

Introduction Planar graphs are subject of interest not only in graph theory [7, 6], but also in defining syntaxes for monoidal categories [10] with specific topological properties, for example diagrams for quantum circuits where crossing wires is a non-trivial operation [4]. We present work on an intrinsically typed data structure of planar graphs, implemented in Agda.
A graph is planar if it can be drawn on a sphere without any edges crossing. We are working at the level of the drawing, so graphs are actually plane graph embeddings. A graph embedding is uniquely defined by an edge ordering around each of its vertices, called a rotation system [5].

Graphs are Decorated Trees The graphs we describe are connected, and may contain self-loops at a vertex as well as multiple edges in parallel. We define graphs inductively by using one of their spanning trees as a skeleton and storing the remaining edges alongside [2]. Our graphs have labelled edges, but also contain data in *sectors*, which are regions near vertices, subdivided by their incident edges. Sectors are also the places within a graph we can point at, or move to. The example on the right shows the spanning tree of the graph in blue, and its sectors as boxes, with a specially marked *root* sector. The data structure represents the traversal of a graph in order, starting from the root, working clockwise round the spanning tree, following tree edges and passing non-tree edges on the way. Arrowheads indicate the order of traversal (the graph itself is undirected).



At each `Step` in the traversal we either encounter the next `sector` or an edge; these two always alternate. When visiting a tree edge we ask for the subtree attached to it. The first time we meet each non-tree edge, we push its label onto a *stack*, popping when we find its other end. A `Step` is indexed by the stack before and after, as well as an indicator of whether we expect a `sector` (of type `S`) or an `edge` (of type `E`) at our next encounter. The stack is a list (with `,-` being cons), and `span`ning subtrees are the reflexive transitive closure `Star` of the `Step` relation.

```
data Step : (List E × SE) → (List E × SE) → Set where
  sector : S → Step (es , sec)(es , edg)
  push   : (e : E) → Step (     es , edg)(e ,- es , sec)
  pop    : (e : E) → Step (e ,- es , edg)(     es , sec)
  span   : E → V → Star Step (as , sec)(bs , edg)
                 →      Step (as , edg)(bs , sec)
```

All together, a plane graph consists of a vertex and a `Star Step ([] , sec)([] , sec)`: a clockwise traversal around that vertex, starting from an empty stack and expecting the root sector, visiting the rest of the tree, then finishing with an empty stack back at the root sector.

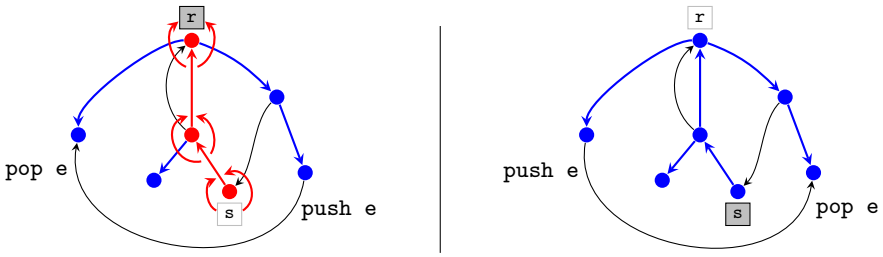
Proposition 1. *A spanning tree together with a stack of additional edges defines a plane graph.*

Proof Sketch. A spanning tree is plane by definition, thus the topological property is established by the treatment of non-tree edges. By maintaining the stack in order of traversal, edges are

popped in reverse order to being pushed, so no two edges cross. Effectively, we contract the spanning tree to a single vertex with only self-loops, i.e. the non-tree edges. The graph is planar when these self-loops form a well bracketed word [2]. □

There is more information yet in the stack indices: each edge e is boundary to a region of the graph. Anything on top of e on the stack is *local* to the region, and once we have popped e we know we have left the region. Observing the stack alone gives clues about where we currently are in the graph, and which edges are local to the current subgraph.

Zippers The first graph operation we want is focussing to a specific position [1], then moving the graph's root to this position. We use zippers [8] for graphs in a bifunctor representation [9], allowing us to transform sector data as we traverse clockwise.
Standard zippers for trees construct a path through the structure by successively choosing one branch to move along, while storing its siblings alongside. In the case of graphs, siblings could be trees but also stack operations. Each layer in the path stores a forwards list of steps ahead in the tree and a backwards list of steps behind (i.e. between the arrival point and the current position). The zipper itself is a sequence of layers surrounding the sector in focus. The left example shows in red the path from sector `s` back to the root. At each vertex along the path the curved arrows depict the backwards and forwards sibling lists:



Re-rooting a graph to a zipper's sector-in-focus involves rotating the traversal order of the spanning tree while keeping track of the direction of the non-tree edges.

Proposition 2. *Re-rooting a planar graph returns a planar graph.*

Proof Sketch. The spanning tree is traversed in a different order, but structurally unchanged. Some of the additional edges have to be turned in the process of moving the root. In the original graph, these edges were pushed before we arrive at the new root, and popped after, they were exactly the index of the new root. When re-rooting, each of these edges will change its direction and the order of edges on the stack will be reversed, thus planarity is maintained. □

In the left example, the stack operations for edge `e` are explicitly marked. The stack at segment `s` is `(e ,- [])`. The right example show the result of the re-rooting operation, with `s` now being the root (with index `[]`), and the stack operations of `e` interchanged.

A Context Comonad We define graphs as containers. Sectors are places for data, as well as places to view the graph from. We obtain a context comonad [11] whose counit projects the root sector data and whose comultiplication *decorates* each sector with the graph re-rooted to that sector. The graph stays the same, with the order of edges around vertices fixed, but we redirect its spanning tree, depending on which sector is root. In future work we intend to decorate each push and pop with the graph obtained by following their non-tree edge, allowing convenient but read-only graph traversal [3].

References

[1] Michael Gordon Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. ∂ for data: Differentiating data structures. *Fundam. Informaticae*, 65(1-2):1–28, 2005.

[2] Olivier Bernardi. Bijective counting of tree-rooted maps and shuffles of parenthesis systems, 2006.

[3] Richard S. Bird. On building cyclic and shared structures in haskell. *Formal Aspects Comput.*, 24(4-6):609–621, 2012.

[4] Bob Coecke, Ross Duncan, Aleks Kissinger, and Quanlong Wang. *Generalised Compositional Theories and Diagrammatic Reasoning*, pages 309–366. Springer Netherlands, Dordrecht, 2016.

[5] John Robert Edmonds Jr. *A combinatorial representation for oriented polyhedral surfaces.* PhD thesis, 1960.

[6] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[7] Jonathan L Gross and Thomas W Tucker. *Topological graph theory.* Courier Corporation, 2001.

[8] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[9] Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 287–295. ACM, 2008.

[10] Peter Selinger. A survey of graphical languages for monoidal categories. *Lecture Notes in Physics*, page 289–355, 2010.

[11] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. In Jirí Adámek and Clemens Kupke, editors, *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science, CMCS 2008, Budapest, Hungary, April 4-6, 2008*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 263–284. Elsevier, 2008.

A Domain-Specific Language for Name Modifiers

Kuen-Bang Hou (Favonia)[1]

University of Minnesota, Minneapolis, Minnesota, U.S.A
kbh@umn.edu

Abstract

Name management is an important factor of usability, but is often an overlooked aspect of the design of programming languages. The issue becomes even more pressing in the case of proof assistants, for a proof frequently depends on different definitions or properties of conflicting names. While names are often not part of the core type theory that a proof assistant is based upon, effective management of them is one decisive factor of practical usability. I therefore designed a compositional and extensible domain-specific language for manipulating hierarchical names. The language has been implemented as a standalone OCaml library and is used in the proof assistants cooltt and algaett.

1 Introduction

Software engineering practice encourages dividing a large program into multiple units, enabling separate or incremental compilation. The same principle applies to any serious mechanized proof using dependent type theory, as type-checking often takes significant time. The efficiency of compilation or type-checking is not the only reason—the division also serve as an opportunity to organize definitions and lemmas according to their conceptual closeness.

In order to support the use of multiple units, most programming languages provide "import" or "include" mechanisms for a unit to access content in another. Those mechanisms bring in declarations from another unit so that local code can access them as if they were defined locally. However, such an "import" feature introduces a new problem—the imported content might shadow existing content in the local scope. As a result, many languages allow programmers to rename, select, or hide parts of imported content. A common strategy is to place new content in a namespace, which can be understood as a special case of group renaming.

The interplay between these mechanisms—renaming, selection, hiding, namespaces, and others—is unfortunately unclear in most designs. For example, consider the Agda statement

```
open A using (x) renaming (x to y) hiding (y)
```

Would Agda make what's named `x` available as both `x` and `y`, or only as `y`, or perhaps unavailable? The above statement would actually be rejected because of the ambiguity. Arguably, how renaming, selection, and hiding could work together is unclear from the concrete syntax.

The core issue is that *name modifiers are effectful*. Most designs chose to limit the syntax to avoid confusing interactions. The other approach, which I believe is more fruitful, is to design a proper domain-specific language with powerful combinators such as sequencing, union, and scoping. Such a language gives programmers the full power to manipulate names.

Another motivation to redesign how names work is to facilitate patching an API for compatibility. Often, a minor update of a library only introduces new bindings, and a client can start using the new API while supporting old ones by implementing those bindings during the transition. For example, suppose `is_prefix` was recently introduced in the `string` namespace in the standard library. The client should be able to inject its own implementation of `is_prefix` into `string` to support older versions of the standard library while updating its codebase.

The desire to patch libraries motivates an *implicit* treatment of namespaces, which means a namespace is only a group of names that happen to share the same prefix. The ML-style module system, on the other hand, limits the ability to change components of a module to make sure it always has a valid signature. There has been work on liberating ML-style modules [3], but the untyped nature of namespaces means we are not subject to these constraints and can derive a simpler design. Moreover, in the context of proof assistants, one should recognize top-level definitions might not have an (internalized) type anyway. It is perhaps easier and better to support both untyped namespaces and typed modules (or records in dependent type theory).

2 The Design

The effectfulness of name modifiers and the need to patch libraries suggest that all operations should work on names sharing a prefix (that is, a subtree) instead of individual names. For example, selection of `a.b` should be understood as selecting all names with the prefix `a.b`. Other important considerations include detection of typos, decoupling from export control, expressiveness, extensibility with custom hooks, and conciseness of the core language. Balancing out these factors led to a language with the following six operators:

- **Emptiness-Checking:** err if there are no matching bindings; this is to detect typos.
- **Scoping:** apply a modifier to names with a given prefix.
- **Renaming:** change names with a given prefix to names with the new prefix.
- **Sequencing:** apply a list of modifiers in order.
- **Union:** take the union of the results of a list of modifiers.
- **Hook:** apply a custom hook.

As a demonstration of expressiveness, the modifier that hides names with the prefix `a.b` can be implemented as a combination of **Scoping** (to focus on the names with the prefix `a.b`), **Emptiness-Checking** (to detect typos), and then **Union** of an empty list (to drops all bindings). Therefore, we do not need a modifier dedicated to hiding in the core language.

3 Discussions

The language has been implemented as a standalone OCaml library named Yuujinchou [1]. It was used in the experimental proof assistants cooltt and algaett. Racket is one of the few languages that provide an equally powerful language for modifying names, and the language has specialized operators for different phases in Racket [2]. However, its support of hierarchical names seems to be limited because its renaming and prefixing modifiers (`rename-in` and `prefix-in`) cannot directly place bindings under a namespace. Other than Racket, most programming languages and proof assistants give little power to its users, and perhaps our simple yet compositional domain-specific language could inspire.

References

[1] Kuen-Bang Hou (Favonia). Yuujinchou: Name modifiers. http://www.github.com/RedPRL/yuujinchou.

[2] Matthew Flatt and PLT. Importing and exporting: require and provide. https://docs.racket-lang.org/reference/require.html.

[3] Norman Ramsey, Kathleen Fisher, and Paul Govereau. An expressive language of signatures. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, page 27–40, New York, NY, USA, 2005. Association for Computing Machinery.

A Flexible Multimodal Proof Assistant

Philipp Stassen, Daniel Gratzer, and Lars Birkedal

Aarhus University
stassen@cs.au.dk, gratzer@cs.au.dk, birkedal@cs.au.dk

A fundamental benefit to working type theoretically is the possibility of working within a proof assistant, which can check and even aid in the construction of complex theorems. Implementing a proof assistant, however, is a highly nontrivial task. In addition to a solid theoretical foundation for the particular type theory, numerous practical implementation issues must be addressed.

Recently, there has been a growing interest in type theories which include *modalities*, unary type constructors which need not commute with substitution. While there has been rapid progress on modal type theories, it is unknown whether standard implementation techniques extend to them. Mainstream proof assistants have begun to experiment with modalities [Vez18], but these implementations are costly and must be specialized to a particular modal situation. Realistically, a practitioner may use a collection of modalities for only one proof and it is impractical to invest in a specialized proof assistant each time. Similar churn has afflicted modal type theories generally and it has pushed type theorists to define *general* systems which can be instantiated with various modalities [Gra+20; LSR17]. We believe that a similar approach alleviate the constant need to implement new modal proof assistants.

Here we focus on MTT [Gra+21], a general modal type theory which can internalize arbitrary collections of (dependent) right adjoints [Bir+20]. These modalities are specified by mode theories [LS16], 2-categories whose objects correspond to modes, morphisms to modalities, and 2-cells to natural transformations between modalities.

MTT has been used to model a variety of existing modal type theories including calculi for guarded recursion, internalized parametricity, and axiomatic cohesion. Better still, MTT has a robustly developed metatheory [Gra+21] which applies *irrespective* of the chosen modalities. An implementation of MTT could therefore conceivably be designed to allow the user to freely change the mode theory without reimplementing the entire proof assistant each time. In fact, recently Gratzer [Gra22] has proven a generic normalization result for MTT. While this result provides the foundation of a flexible implementation, it remains to convert this proof into a practical type checking algorithm.

From theory to practice Converting the theoretical guarantee of normalization into an executable program is not a small step. A first obstacle is the syntax of MTT itself: prior work has exclusively considered an algebraic presentation of the syntax as a generalized algebraic theory. While mathematically elegant, a proof assistant requires a more streamlined and ergonomic syntax. Once a more convenient syntax has been designed, one must adapt the normalization proof to a normalization algorithm. Normalization is proven by a sophisticated *gluing* argument, and while the proof is reminiscent of normalization-by-evaluation [Abe13] it remains to extract such an algorithm.

The majority of subtleties in the implementation flow not from the modal types *per se*, but from the extensions to contexts and substitutions needed to support them. In particular, each 2-cell in the mode theory induces a new form of substitution and these *key substitutions* accumulate at variables. As a result, each variable in MTT is annotated with a 2-cell describing its extraction from the context. This additional piece of data disrupts a crucial property of modern

NbE: variables can no longer be represented in a way invariant under weakening. Without this invariant, one cannot adapt the substitution-free NbE familiar from MLTT [Abe13].

We therefore only consider the restricted class of *preordered* mode theories—mode theories with at most one 2-cell between any pair of modalities. In this case, the modal substitution apparatus for MTT is dramatically simplified, and the problematic substitutions evaporate. Without this obstruction, we have successfully implemented a normalization algorithm which elegantly handles multiple modalities. Crucially, the normalization algorithm does not depend on the particulars of the mode theory and can be applied without change to any preordered collection of modalities.

Normalization-by-evaluation The normalization proof for MTT follows the structure of a normalization-by-evaluation proof. Rather than fixing a rewriting system, a term is *evaluated* into a carefully chosen semantics equipped with a quotation function reifying an element of the semantic domain to a normal form. The entire normalization function is then a round-trip from syntax to semantics and then back to normal forms. While the proof of normalization uses a traditional denotational model for a semantic domain, this is impractical for implementation.

Instead `mitten` follows the literature on normalization-by-evaluation and uses a *defunctionalized* and *syntactic* semantic domain [Abe13]. This approach has previously been adapted to work with a modal type theory [GSB19], but adapting it to MTT introduces several novel challenges. In particular, variables must be annotated with 2-cells from the mode theory, and managing these annotations ripples out through the entire algorithm. We show that for preordered mode theories said annotations omitted and reconstructed during type checking.

Mode theories Normalization for MTT does not immediately imply the decidability of type equality. Terms (and therefore types) mention both 1- and 2-cells from the mode theory, and deciding their equality is a necessary precondition for deciding type equality. Moreover, deciding the equality of 1- and 2-cells, even in a finitely presented 2-category, is well-known to be undecidable.[1] Special attention is therefore necessary for each mode theory to ensure that the normalization algorithm for MTT is sufficient to yield a type-checker. While this rules out a truly generic proof assistant for MTT which works regardless of the choice of mode theory, `mitten` shows that the next best result is obtainable. We implement `mitten` parametrically in a module fully describing the mode theory and show that the type-checker can be designed to rely only on the existence of such a decision procedure.

Contributions We contribute a defunctionalized NbE algorithm which reduces the type checking problem for MTT to deciding the word problem for the mode theory.

Furthermore, we specify a bidirectional syntax for MTT together with a type checking algorithm. Type checking is decidable if and only if the word problem of the mode theory is.

We have put these results into practice with `mitten`, an prototype implementation of MTT based on this algorithm. `mitten` supports the replacement of the underlying mode theory with minimal alterations, allowing a user to construct specialized proof assistants for modal type theories by merely supplying a single module specifying the mode theory together with equality functions for 0-, 1-, and 2-cells.

We presented a version of this ongoing work at the WITS workshop in January 2022.

[1] The word problem is well-known to be undecidable for finitely presented groups which can be realized as finitely-presented categories and therefore locally discrete finitely-presentable 2-categories.

References

[Abe13] Andreas Abel. "Normalization by Evaluation: Dependent Types and Impredicativity". Habilitation. Ludwig-Maximilians-Universität München, 2013 (cit. on pp. 1, 2).

[Bir+20] Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. "Modal dependent type theory and dependent right adjoints". In: *Mathematical Structures in Computer Science* 30.2 (2020), pp. 118–138. DOI: 10.1017/S0960129519000197. eprint: 1804.05236 (cit. on p. 1).

[Gra22] Daniel Gratzer. "Normalization for multimodal type theory". In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3531130.3532398 (cit. on p. 1).

[Gra+21] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. "Multimodal Dependent Type Theory". In: *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021). DOI: 10.46298/lmcs-17(3:11)2021. URL: https://lmcs.episciences.org/7713 (cit. on p. 1).

[Gra+20] Daniel Gratzer, G.A. Kavvos, Andreas Nuyts, and Lars Birkedal. "Multimodal Dependent Type Theory". In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '20. ACM, 2020. DOI: 10.1145/3373718.3394736 (cit. on p. 1).

[GSB19] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. "Implementing a Modal Dependent Type Theory". In: *Proc. ACM Program. Lang.* 3 (ICFP 2019). DOI: 10.1145/3341711 (cit. on p. 2).

[LS16] Daniel R. Licata and Michael Shulman. "Adjoint Logic with a 2-Category of Modes". In: *Logical Foundations of Computer Science*. Ed. by Sergei Artemov and Anil Nerode. Springer International Publishing, 2016, pp. 219–235. DOI: 10.1007/978-3-319-27683-0_16 (cit. on p. 1).

[LSR17] Daniel R. Licata, Michael Shulman, and Mitchell Riley. "A Fibrational Framework for Substructural and Modal Logics". In: *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*. Ed. by Dale Miller. Vol. 84. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 25:1–25:22. DOI: 10.4230/LIPIcs.FSCD.2017.25 (cit. on p. 1).

[Vez18] Andrea Vezzosi. *agda-flat*. 2018. URL: https://github.com/agda/agda/tree/flat (cit. on p. 1).

A Generalized Translation of Pure Type Systems

Nathan Mull

University of Chicago
Chicago, Illinois, U.S.A.
nmull@uchicago.edu

Introduction. The class of pure type systems was introduced by Terlouw [11] and Berardi [4] (and further developed by Barendregt [1, 2]) as a natural generalization of the lambda cube; it contains the lambda cube as well as systems with richer sort structure and quantification. Formally, a pure type system (PTS) λS is specified by a set of sorts $\mathcal{S}$, a set of axioms $\mathcal{A}$ satisfying $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ and a set of rules $\mathcal{R}$ satisfying $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$, and has the same derivation rules as those given for the lambda cube except in the case of axioms and Π-type formation, which are replaced respectively with the following:

$$\vdash_{\lambda S} s_1 : s_2 \qquad\qquad \frac{\Gamma \vdash_{\lambda S} A : s_1 \qquad \Gamma, x : A \vdash_{\lambda S} B : s_2}{\Gamma \vdash_{\lambda S} \Pi x^A B : s_3}$$

where $(s_1, s_2) \in \mathcal{A}$ and $(s_1, s_2, s_3) \in \mathcal{R}$.

The study of pure type systems can be viewed as the study of how sort structure affects the meta-theoretic properties of a type system, especially because of the minimal set of type formers (*e.g.*, there are no Σ-types by default). One such meta-theoretic property, arguably one of the most important, is normalization. A type system is *weakly normalizing* if every typable term has a normal form and is *strongly normalizing* if no typable term appears in an infinite reduction sequence. Girard [6] demonstrated that sort structure can have a nontrivial effect on the normalization behavior of a type system by showing that the PTS λU is not strongly normalizing. In particular, circularity in the sort structure of a pure type system (λU does not explicitly include an axiom of the form "*Type* is a *Type*") is not a necessary condition for non-normalization. This leaves open a fundamental question: what is the relationship between the sort structure and normalization?

The last few decades have seen numerous techniques for proving normalization of systems in the lambda cube and their extensions, and some of these techniques have been extended to the PTS setting (*e.g.*, Melliès and Werner [9] extend the notion of Λ-sets to pure type systems) but many have not. The purpose of this abstract is to outline the generalization of one such technique, which might be called *dependency eliminating translations*.

Contributions. One approach for proving strong normalization of a type system is to define a typability-preserving infinite-reduction-path-preserving translation from that system into a weaker system which is already known to be strongly normalizing. Harper et al. [7] define such a translation from λP to $\lambda_{\rightarrow}$ and Geuvers and Nederhof [5] extend that translation to one from λC to $\lambda\omega$. Both of these translations can be viewed as deleting the *dependent* rule in the corresponding system, *i.e.*, the rule $(*, \Box)$, which allows types to depend on terms. For sufficiently well-structured pure type systems, this notion of dependence can be generalized, as is done by Barthe et al. [3] for their definition of *generalized non-dependent* pure type systems. I extend these translations to pure type systems in a way that maintains the property that dependent rules are deleted.

Before stating the following theorem, a few definitions. A PTS is *n-tiered* if it is of the form

$$\begin{aligned}\mathcal{S} &= \{s_i \mid i \in [n]\}\\ \mathcal{A} &= \{(s_i, s_{i+1}) \mid i \in [n-1]\}\\ \mathcal{R} &\subset \{(s, s', s') \mid (s, s') \in \mathcal{S} \times \mathcal{S}\}\end{aligned}$$

A tiered PTS is (i,j)*-full* if its rules contain $\{(s_l, s_k, s_k) \mid l \leq i \text{ and } l \leq k \leq j\}$ and is *full* if it satisfies the following closure property: if $(s_i, s_j, s_j) \in \mathcal{R}_{\lambda S}$ then λS is (j,i)-full. For any tiered PTS λS, define its *non-dependent restriction*, denoted λS^*, to be the tiered system with rules $\{(s_i, s_j, s_j) \in \mathcal{R}_{\lambda S} \mid i \leq j\}$.

Theorem 1. *For any full tiered PTS λS, there are two functions $\tau : \mathsf{T} \to \mathsf{T}$ and $[\![-]\!] : \mathsf{T} \to \mathsf{T}$ on terms such that the following hold.*

1. *If $\Gamma \vdash_{\lambda S} A : B$ then there is a context Γ' such that $\Gamma' \vdash_{\lambda S^*} [\![A]\!] : \tau(B)$. That is, $[\![-]\!]$ preserves typability.*
2. *For any term A derivable in λS, if $A \twoheadrightarrow_\beta B$, then $[\![A]\!] \twoheadrightarrow_\beta^+ [\![B]\!]$. That is, $[\![-]\!]$ preserves infinite reduction paths.*

This implies the strong normalization of any full tiered PTS depends only on the strong normalization of its non-dependent restriction. Unfortunately, fullness is a very strong property. A system which is (i,j)-full where $i \geq 2$ and $j \geq 3$, for example, contains λU and is thus inconsistent. There is only a small class of systems on which the translation can be applied non-trivially, and every system in this class is a subsystems of Lou's extended calculus of constructions (ECC) [8], which is known to be strongly normalizing. The strongest of these full n-tiered system has the rules

$$\{(s_k, s_1, s_1) \mid k \in [n]\} \cup \{(s_1, s_k, s_k) \mid k \in [n]\} \cup \{(s_2, s_k, s_k) \mid k \in [n]\}$$

Together with a proof that the non-dependent restriction of this system is strongly normalizing, we get a modular proof that this system is strongly normalizing along the lines of the Geuvers-Nederhof result. In particular, this proof does not require a detour through quasi-normalization (as is done by Luo for ECC) and the system for which one ultimately has to prove strong normalization after translation (by, say, the Girard-Tait method) is simpler.

Discussion. The restriction to tiered systems is for convenience, and it turns out to be sufficient even if we want to consider more general classes along the lines of persistent, stratified systems (see [3]) because such systems can be viewed as disjoint unions of tiered systems, in the sense of [10]. However, the restriction of fullness is clearly quite limiting. The generalization is, I believe, a fairly faithful one, so it seems possible that a more sophisticated translation could push this idea further. Being able to handle even just one additional non-dependent rule could be advantageous. For example, I became interested in translations like this one because of their potential application to the Barendregt-Guevers-Klop conjecture, an open question which posits that weak normalization implies strong normalization for all pure types systems. Barthe et al. [3] prove the conjecture holds for a class of non-dependent pure types systems via a CPS-style translation. If the non-dependent restriction of a tiered PTS is captured by the conditions of their theorem then their result can be leveraged and extended by a very simple bootstrapping argument: since weak normalization of λS implies weak normalization of λS^*, if λS is weakly normalizing then, in fact, λS^* is strongly normalizing, which implies λS is as well.

References

[1] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[2] Henk Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science, Volume II*, pages 117–309. Oxford University Press, 1993.

[3] Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. Weak normalization implies strong normalization in a class of non-dependent pure type systems. *Theoretical Computer Science*, 269(1-2):317–361, 2001.

[4] Stefano Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Carnegie Mellon University, Universita di Torino, 1988.

[5] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[6] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993.

[8] Zhaohui Luo. *An extended calculus of constructions*. PhD thesis, University of Edinburgh, 1990.

[9] Paul-André Mellies and Benjamin Werner. A Generic Normalisation Proof for Pure Type Systems. In *International Workshop on Types for Proofs and Programs*, pages 254–276. Springer, 1996.

[10] Cody Roux and Floris van Doorn. The Structural Theory of Pure Type Systems. In *Rewriting and Typed Lambda Calculi*, pages 364–378. Springer, 2014.

[11] Jan Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmege, 1989.

A Reasonably Gradual Type Theory

Kenji Maillard[1], Meven Lennon-Bertrand[1], Nicolas Tabareau[1], and Éric Tanter[2]

[1] Gallinette Project-Team, Inria, Nantes, France
[2] PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile

Abstract

Gradualizing the Calculus of Inductive Constructions (CIC) involves dealing with subtle tensions between normalization, graduality, and conservativity with respect to CIC [5]. We present GRIP, a novel approach to gradual dependent types combining an impure sort of types inhabited by type casts, unknown terms and errors, as well as a pure sort of strict propositions internalizing precision. Internal precision supports reasoning about graduality within GRIP itself, for instance to characterize gradual exception-handling terms, and supports gradual subset types. The metatheory of GRIP is supported by a model formalized in Coq, and we provide a prototype implementation in Agda.[1]

Extending gradual typing [13, 14] to dependent types is a challenging endeavor due to the intricacies of type checking and conversion in presence of imprecision at both the type and term levels. While early efforts looked at gradualizing specific aspects of a dependent type system (*e.g.,* subset types and refinements [4, 15], or the fragment without inductive types [2]), we recently studied gradual typing in the context of the Calculus of Inductive Constructions (CIC) [5], the theory at the core of many proof assistants such as Coq [16] and Agda [8, 9]. There, we introduced GCIC, a gradual source language, whose semantics is given by elaboration to a dependently-typed calculus, called CastCIC.

A Dependently-Typed Cast Calculus CastCIC is an extension of Martin-Löf type theory (MLTT) [6] with (non-indexed) inductive types, and with exceptions as introduced by [10]. For a given type A, there are two exceptional terms, namely $\mathtt{err}_A$ representing runtime type errors, and $?_A$ representing the *unknown term*, which can optimistically stand for any term of type A. Additionally, CastCIC features a cast operator $\langle B \Leftarrow A \rangle\, t$, which supports treating a term t of type A as a term of type B, without requiring any relation between A and B. Intuitively, the cast operator is the identity when A and B are convertible, and fails when A and B are incompatible, for instance when A is the type of natural number and B is a function type. This intuition is made explicit by the notion of *(im)precision*: when a type A is more precise than B, written $A \sqsubseteq B$, then casting from A to B does not fail, and doing the roundtrip back to A is the identity; the formal formulation of this property, coined *graduality* by [7], is that when $A \sqsubseteq B$, the cast operations induce an embedding-projection pair between A and B. Additionally, $?$ is the least precise type, and therefore casting from A to the unknown type $?$ and back is always the identity. We previously uncovered in [5] an inherent tension which states that three fundamentally desirable properties cannot be fully satisfied simultaneously: (1) strong normalization, a property of particular relevance in the context of proof assistants, (2) conservativity with respect to CIC, namely the ability to faithfully embed the static theory in the gradual theory,[2] and (3) graduality, which guarantees that typing and evaluation are monotone with respect to precision. The maximality of the unknown type is a key element of this tension. Indeed, if $? \to ? \sqsubseteq ?$, then by graduality it is possible to embed the untyped lambda calculus, and in particular the diverging term $\Omega := (\lambda\, x : ?.\ x\ x)\ (\lambda\, x : ?.\ x\ x)$.

[1] https://gitlab.inria.fr/kmaillar/grip-a-reasonably-gradual-type-theory
[2] Not to be confused with logical conservativity!

Relaxing ?'s Maximality In this work, we observe that an adequate stratification of precision enables a full account of graduality for an extension of CastCIC, called GRIP. The key idea is that $?_{\square_i}$ should be the least precise type among all types at level i and below, *except* for dependent function types at level i (which are however still less precise than $?_{\square_{i+1}}$). We can precisely characterize problematic terms as those that are not *self-precise* (*i.e.,* more precise than themselves), which for function types means non-monotone with respect to precision. The prototypical example of a non-monotone term is that of recursive large elimination, such as the type of n-ary functions over natural numbers (in Coq):

```
Fixpoint nArrow (n : ℕ) : □₀ := match n with 0 ⇒ ℕ | S m ⇒ ℕ → narrow m.
```

The term `nArrow n` is a type (*i.e.,* a term of type $\square_0$), and we have for example `nArrow` $0 \equiv \mathbb{N}$ and `nArrow` $2 \equiv \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. `nArrow` is not monotone because, given $n \sqsubseteq ?_{\mathbb{N}}$, there is no fixed level i for which `nArrow` $n \sqsubseteq ?_{\square_i}$ for any n. Another more practical example is that of a dependently-typed `printf` function, whose actual arity depends on the input string. We prove that the dynamic gradual guarantee holds in GRIP for any self-precise context, and that casts between types related by precision induce embedding-projection pairs between self-precise terms. Therefore, this shift in perspective in the interpretation of the unknown type and the associated notion of precision yields a gradual theory that conservatively extends CIC, is normalizing, and satisfies graduality for a large and well-defined class of terms.

Internalizing Precision, Reasonably While we could study graduality for GRIP externally, we observe that we can exploit the expressiveness of the type-theoretic setting to internalize precision and its associated reasoning. In particular this makes it possible to state and prove, within the theory itself, results about (self-)precision and graduality for specific terms. Internalizing precision requires solving an important obstacle: when adding exceptions to MLTT [10], the theory becomes inconsistent as a logic, because it is possible to inhabit any type A by raising an exception $\mathtt{err}_A$. In the gradual setting, there is also the alternative of using the unknown term $?_A$ to inhabit any type A, so we need to avoid these degenerate proofs and provide a logically consistent theory. Moreover, useful reasoning on a notion of precision as error approximation [7] requires support from the gradual type theory in the shape of extensionality principles, an established challenge inside intensional type theories such as MLTT or CIC.

We address both issues by combining recent advances in type theory: the reasonably exceptional type theory RETT [11], that supports consistent reasoning about exceptional terms, and the observational type theory $\mathsf{TT}^{\mathsf{obs}}$ [12] that provides a setoidal equality in a specific universe $\mathbb{P}$ of definitionally proof-irrelevant propositions. A major insight of this work is to realize that we can actually merge the logical universe of RETT used to reason about exceptional terms with the universe $\mathbb{P}$ of proof-irrelevant propositions in order to define an internal notion of precision that is extensional and whose proofs cannot be trivialized with exceptional terms. We support this claim by formalizing a model in Coq using partial preorders (Footnote 1).

Applications of Internal Precision In addition to supporting reasoning about the graduality of terms in a theory that is not globally gradual, internal precision makes it possible to support gradual subset types, in which a type can be refined by a proposition expressed using precision. Moreover, in the literature, exception handling is never considered when proving graduality because this mechanism inherently allows terms that do not behave monotonically with respect to precision. Internal precision enables us to support exception handling in the impure layer of the type theory, and to consistently reason about the graduality (or not) of exception-handling terms. We illustrate these examples in a proof-of-concept implementation in Agda using rewrite rules [1] to evaluate the design of GRIP (Footnote 1).

References

[1] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proceedings of the ACM on Programming Languages*, 2020.

[2] Joseph Eremondi, Éric Tanter, and Ronald Garcia. Approximate normalization for gradual dependent types. In ICFP 2019 [3], pages 88:1–88:30.

[3] *Proceedings of the 24th ACM SIGPLAN Conference on Functional Programming (ICFP 2019)*, volume 3. ACM Press, August 2019.

[4] Nico Lehmann and Éric Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*, pages 775–788, Paris, France, January 2017. ACM Press.

[5] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. Gradualizing the calculus of inductive constructions. *ACM Transactions on Programming Languages and Systems*, 44(2), June 2022.

[6] Per Martin-Löf. An intuitionistic theory of types, 1971. Unpublished manuscript.

[7] Max S. New and Amal Ahmed. Graduality from embedding-projection pairs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Functional Programming (ICFP 2018)*, volume 2, pages 73:1–73:30. ACM Press, September 2018.

[8] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[9] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming (AFP 2008)*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2009.

[10] Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option - an exceptional type theory. In Amal Ahmed, editor, *Proceedings of the 27th European Symposium on Programming Languages and Systems (ESOP 2018)*, volume 10801 of *Lecture Notes in Computer Science*, pages 245–271, Thessaloniki, Greece, April 2018. Springer-Verlag.

[11] Pierre-Marie Pédrot, Nicolas Tabareau, Hans Fehrmann, and Éric Tanter. A reasonably exceptional type theory. In ICFP 2019 [3], pages 108:1–108:29.

[12] Loïc Pujet and Nicolas Tabareau. Observational Equality: Now For Good. *Proceedings of the ACM on Programming Languages*, 6(POPL), January 2022.

[13] Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[14] 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, Asilomar, California, USA, May 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] Éric Tanter and Nicolas Tabareau. Gradual certified programming in Coq. In *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*, pages 26–40, Pittsburgh, PA, USA, October 2015. ACM Press.

[16] The Coq Development Team. *The Coq proof assistant reference manual*. 2020. Version 8.12.

A Simple Concurrent Lambda Calculus for Session Types

Jules Jacobs

Radboud University Nijmegen, mail@julesjacobs.com

Abstract

We introduce μGV ("micro GV"), which strives to be a minimal extension of linear λ-calculus with concurrent communication, adding only a new **fork** construct for spawning threads. The child and parent thread communicate with each other via two dual values of linear function type $\tau_1 \xrightarrow{\text{lin}} \tau_2$ and $\tau_2 \xrightarrow{\text{lin}} \tau_1$. Using only **fork**, we can implement all of GV's channel operations and session types as a library in μGV. The linear type system ensures that μGV programs are deadlock-free and satisfy global progress, which we prove in Coq.

Session types [7, 6] for communication channels can be used to verify that programs follow the protocol specified by a channel's session type. Gay and Vasconcelos [5] embed session types in a linear λ-calculus with concurrency and channels, and Wadler's subsequent GV [13] and its derivatives [10, 11, 12, 4, 3] guarantee that all well-typed programs are deadlock free.

To add session types to linear λ-calculus, one adds session type formers and their corresponding channel operations: $\mathbf{!}\tau.s$ (send a message of type τ, continue with protocol s), $\mathbf{?}\tau.s$ (receive a message of type τ, continue with s), $s_1 \oplus s_2$ (send choice between s_1 and s_2), $s_1 \,\&\, s_2$ (receive choice between s_1 and s_2), and End (close channel). An example is $\mathbf{!}\tau_1.(\mathbf{?}\tau_2.\mathsf{End} \oplus \mathbf{!}\tau_3.\mathsf{End})$: send a value of type τ_1 then either receive τ_2 or send τ_3. One adds **fork** for creating a thread and a pair of dual channel endpoints for communication between the parent and child thread. For this, one needs a definition of duality, with **!** dual to **?**, $\oplus$ dual to &, and End dual to itself.

μGV, on the other hand, has none of these. Instead, we add only a *single* construct: **fork**.

$$\mathbf{fork} : ((\tau_1 \xrightarrow{\text{lin}} \tau_2) \xrightarrow{\text{lin}} \mathbf{1}) \to (\tau_2 \xrightarrow{\text{lin}} \tau_1)$$

μGV adds no new type formers, and no explicit definition of duality. Instead, we re-use the linear function type $\tau_1 \xrightarrow{\text{lin}} \tau_2$ for communication between threads. Let us look at an example:

$$\mathbf{let}\ c = \mathbf{fork}(\lambda c'.\ \mathbf{print}(c'\ 1))\ \mathbf{in}\ \mathbf{print}(1 + c\ 0)$$

This program forks off a new thread and creates *communication barriers* c and c' to communicate between the threads. The barrier c gets returned to the main thread, and c' gets passed to the child thread. A call to a barrier will block until the other side is also trying to synchronize, and will then exchange the values passed as an argument: when $c'\ 1$ is called, it will block until $c\ 0$ is also called, and vice versa. The call $c'\ 1$ will then return 0, and the call $c\ 0$ will return 1. Thus, the program will print 0 2 or 2 0, depending on which thread prints first. Using a tiny channel library, we can write message passing programs:

$$\mathbf{send}(c, x) \triangleq \mathbf{fork}(\lambda c'.\ c\ (c', x)) \qquad \mathbf{receive}(c) \triangleq c\ () \qquad \mathbf{close}(c) \triangleq c\ ()$$

$\mathbf{let}\ x_1 = \mathbf{fork}(\lambda x_1'.\ \mathbf{let}\ (x_2', n_1) = \mathbf{receive}(x_1')$ // receive message n_1
$\qquad\qquad\qquad\quad \mathbf{let}\ (x_3', n_2) = \mathbf{receive}(x_2')$ // receive message n_2
$\qquad\qquad\qquad\quad \mathbf{let}\ x_4' = \mathbf{send}(x_3', n_1 + n_2); \mathbf{close}(x_4'))$ // send $n_1 + n_2$ back and close

$\mathbf{let}\ x_2 = \mathbf{send}(x_1, 1)$ // send message 1

$\mathbf{let}\ x_3 = \mathbf{send}(x_2, 2)$ // send message 2

$\mathbf{let}\ (x_4, n) = \mathbf{receive}(x_3)$ // receive $n = 1 + 2$

$\mathbf{print}(n);\ \mathbf{close}(x_4)$

μGV expressions and types

$$e \in \mathit{Expr} ::= x \mid () \mid (e, e) \mid \mathbf{in_L}(e) \mid \mathbf{in_R}(e) \mid \lambda x.\ e \mid e\ e \mid \mathbf{fork}(e) \mid \mathbf{let}\ (x_1, x_2) = e\ \mathbf{in}\ e \mid \mathbf{match}\ e\ \mathbf{with}\ \ldots\ \mathbf{end}$$

$$\tau \in \mathit{Type} ::= 0 \mid 1 \mid \tau \times \tau \mid \tau + \tau \mid \tau \xrightarrow{\mathsf{lin}} \tau \mid \tau \xrightarrow{\mathsf{unr}} \tau \mid \mu x.\tau \mid x$$

Session types duality

$$\overline{\mathsf{End}} \triangleq \mathsf{End}$$
$$\overline{!\tau.s} \triangleq ?\tau.\overline{s}$$
$$\overline{?\tau.s} \triangleq !\tau.\overline{s}$$
$$\overline{s_1 \oplus s_2} \triangleq \overline{s_1} \,\&\, \overline{s_2}$$
$$\overline{s_1 \,\&\, s_2} \triangleq \overline{s_1} \oplus \overline{s_2}$$

Encoding session types in μGV

$$\mathsf{End} \triangleq \mathbf{1} \xrightarrow{\mathsf{lin}} \mathbf{1}$$
$$!\tau.s \triangleq \overline{s} \times \tau \xrightarrow{\mathsf{lin}} \mathbf{1}$$
$$?\tau.s \triangleq \mathbf{1} \xrightarrow{\mathsf{lin}} s \times \tau$$
$$s_1 \oplus s_2 \triangleq \overline{s_1} + \overline{s_2} \xrightarrow{\mathsf{lin}} \mathbf{1}$$
$$s_1 \,\&\, s_2 \triangleq \mathbf{1} \xrightarrow{\mathsf{lin}} s_1 + s_2$$

Channel operations

$$\mathbf{fork} : (s \xrightarrow{\mathsf{lin}} \mathbf{1}) \to \overline{s}$$
$$\mathbf{close} : \mathsf{End} \to \mathbf{1}$$
$$\mathbf{send} : !\tau.s \times \tau \to s$$
$$\mathbf{receive} : ?\tau.s \to s \times \tau$$
$$\mathbf{tell_L} : s_1 \oplus s_2 \to s_1$$
$$\mathbf{tell_R} : s_1 \oplus s_2 \to s_2$$
$$\mathbf{ask} : s_1 \,\&\, s_2 \to s_1 + s_2$$

Encoding channel operations in μGV

$$\mathbf{fork}(x) \triangleq \mathbf{fork}(x)$$
$$\mathbf{close}(c) \triangleq c\ ()$$
$$\mathbf{send}(c, x) \triangleq \mathbf{fork}(\lambda c'.\ c\ (c', x))$$
$$\mathbf{receive}(c) \triangleq c\ ()$$
$$\mathbf{tell_L}(c) \triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_L}(c'))$$
$$\mathbf{tell_R}(c) \triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_R}(c'))$$
$$\mathbf{ask}(c) \triangleq c\ ()$$

Figure 1: The μGV language (top), session types (left) and their encoding in μGV (right).

As in GV, our channels are used in functional style: each channel operation returns a new channel. This channel will have a new type, reflecting the step in the protocol. In fact, GV's session types (including choice) can be encoded in terms of μGV's types, so that our channel library can be given a statically session-typed interface (see Figure 1). Recursive sessions can be encoded with recursive μGV types.

Like GV, all well-typed μGV programs are automatically *deadlock free*, and therefore satisfy *global progress*. We prove this property in Coq. Because of μGV's simplicity, these proofs are simpler and shorter than previous (mechanized) proofs for deadlock freedom of session types [8], even when counting the encoding of session types into μGV (1442 lines vs 2796 lines).

There have been other efforts for simpler systems, such as an encoding of session types into π-calculus types [2], and *minimal session types* [1], which decompose multi-step session types into single-step session types in a π-calculus (single-shot synchronization primitives have also been used in the implementation of a session-typed channel library for Haskell [9]).

I hope that μGV shows that linear λ-calculus also provides a good substrate for a minimalist concurrent calculus, with communication primitives that capitalize on the fact that the quintessential linear λ-calculus type, the linear function type $\tau_1 \xrightarrow{\mathsf{lin}} \tau_2$, is a *self-dual* connective.

References

[1] Alen Arslanagic, Jorge A. Pérez, and Erik Voogd. Minimal Session Types (Pearl). In *ECOOP 2019*, 2019.

[2] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, 2012.

[3] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. In *CONCUR*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[4] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *PACMPL*, 3(POPL):28:1–28:29, 2019.

[5] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 20(1):19–50, 2010.

[6] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523, 1993.

[7] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138, 1998.

[8] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: A separation logic approach for proving deadlock freedom. In *POPL*, 2022.

[9] Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. Haskell 2021.

[10] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *ESOP*, volume 9032 of *LNCS*, pages 560–584, 2015.

[11] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *ICFP*, pages 434–447, 2016.

[12] Sam Lindley and J. Garrett Morris. Lightweight functional session types. In *Behavioural Types: from Theory to Tools*. 2017.

[13] Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012.

A Theory of Call-by-Value Solvability

Beniamino Accattoli[1] and Giulio Guerrieri[2]

Solvability is a central notion in the semantics of the λ-calculus. First studied by Wadsworth [Wad71, Wad76] and Barendregt [Bar71, Bar74], it characterizes which λ-terms can be seen as producing a result, thus denoting *successful* or *meaningful computations*, and which ones cannot. Instinctively, one would identify results with terms having a normal form, and thus, dually, *unsuccessful* or *meaningless* terms as those not having a normal form. Such an approach has two related drawbacks. Firstly, the representation of partial recursive functions associating *undefinedness* with terms not having a normal form is problematic, as it is not stable by composition. Secondly, the equational theory extending β-conversion $=_\beta$ with the identification of all terms not having a normal form is *inconsistent*, that is, it equates all λ-terms.

The Solvable Theory Both drawbacks disappear if meaningful/meaningless terms are rather identified with solvable/unsolvable terms, where *t is solvable if it exists a head context H sending t to the identity*, that is, such that $H\langle t\rangle \to_\beta^* \mathsf{I}$. Roughly, it means that t can be decomposed by an environment that cannot simply discard t. An important operational characterization is due to Wadsworth [Wad76]: *a term t is solvable if and only if the head reduction of t terminates*. The characterization shows how to refine the naive theory, identifying *meaningful* with *head normalizable* rather than *normalizable*. A compositional representation of partial recursive functions can then be given, and the equational theory extending $=_\beta$ by identifying all unsolvable terms—the *minimal sensible theory* $\mathcal{H}$—is consistent.

Semantics of Call-by-Value Many variants of the λ-calculus have emerged. What is usually referred to as *the* λ-calculus could nowadays be more precisely referred to as the *(strong) call-by-name λ-calculus*. It is the most studied of λ-calculi, and yet it is *never* used in applications. Functional programming languages, in particular, often prefer Plotkin's *call-by-value* λ-calculus [Plo75], where β-redexes can fire only when the argument is a *value* and usually further restrict it to *weak reduction* and to closed terms—what we refer to as *Closed CbV* (λ-calculus).

The denotational semantics of the CbV λ-calculus is much less studied and understood than the CbN one (some notable exceptions are [EHRDR92, PRR99, Ehr12, MPRDR19]). This is not by accident: as first shown by Paolini and Ronchi Della Rocca [PRDR99, Pao01, RP04], there are some inherent complications in trying to adapt semantic notions from CbN to CbV. They stem from the fact that, while Closed CbV is a very elegant setting, denotational semantics have to deal with *open* terms, and Plotkin's operational semantics is not adequate for that because of *premature* normal forms—see Accattoli and Guerrieri for extensive discussions [AG16].

One of the complications is that CbV solvability does not admit an *internal* operational characterization akin to Wadsworth's one for CbN, and thus it is not really manageable.

Two Approaches to Call-by-Value Solvability The literature has explored two opposite approaches towards the difficulty of denotational semantics for the CbV λ-calculus:

1. *Disruptive*: replacing Plotkin's CbV calculus with another, extended CbV calculus as to obtain smoother denotational semantics, and in particular an easier theory of solvability;
2. *Conservative*: considering Plotkin's CbV calculus as untouchable and striving harder to characterize semantical notions (see García-Pérez and Nogueira [GN16]).

One of the achievements of the disruptive approach is an operational characterization of solvability akin to Wadsworth, due to Accattoli and Paolini [AP12]. They introduce a CbV λ-calculus

with let expressions, called *value substitution calculus* (shortened to VSC), together with a *solving reduction* that terminates if and only if the term is solvable in the VSC. This is akin to what happens in CbN, where solvable terms are those for which head reduction terminates.

A reconciliation of the disruptive and the conservative approaches is obtained by Guerrieri et al. [GPR17]. On the one hand, they embrace the disruptive approach, as they study Carraro and Guerrieri's *shuffling calculus* [CG14], another extensions of Plotkin's calculus which can be seen as a variant over the VSC and where Accattoli and Paolini's operational characterization of solvability smoothly transfers. On the other hand, they prove that *a λ-term t is solvable in the shuffling calculus if and only if it is solvable in Plotkin's calculus*. Therefore, the disruptive extension becomes a way to study the conservative notion of solvability for Plotkin's calculus.

Open Questions about CbV Solvability These works paved the way for a theory of CbV solvability analogous to the one in CbN. Such a theory however is still lacking. For instance, it is unknown whether CbV unsolvable terms can be consistently equated, *i.e.*, whether the *minimal sensible theory by value* $\mathcal{H}_{\mathcal{V}}$ is consistent. Further delicate points concern the characterization of CbV solvable terms via intersection types. In CbN, a term t is solvable if and only if t is typable with Gardner-de Carvalho's *non-idempotent* intersection types [Gar94, dC07, dC18], also known as *multi types*. Moreover, one can extract quantitative operational information about solvable terms, namely the number of steps of the head strategy, which is a reasonable measure of time complexity for λ-terms (see Accattoli and Dal Lago [AD12]), as well as the size of the head normal form, as first shown by de Carvalho [dC07, dC18].

In CbV, there exist characterizations of solvable terms via intersection and multi types [PRDR99, KMRDR21]. Those type systems, however, are *defective*: contrarily to what claimed in those papers, their systems do not verify subject reduction (for [PRDR99] subject expansion also fails). Carraro and Guerrieri [CG14] characterize CbV solvability using relational semantics, but their characterization is not purely semantic (or type-theoretic) because it also needs the syntactic notion of *CbV Taylor-Ehrhard expansion* [Ehr12]. Additionally, from none of these characterizations it is possible to extract quantitative operational information. They all rely, indeed, on the shuffling calculus, for which it is unclear how to extract (from type derivations) the number of commuting conversion steps, and its time cost model is also unclear [AG16].

Contributions We study all these questions, providing also a *quantitative* analysis of solvability via intersection types. Because of the quantitative aspect, we study solvability *in the VSC* rather than in the shuffling calculus. The VSC is indeed a better fit than the shuffling calculus for quantitative analyses, because its number of β steps *is* a reasonable time cost model and can be extracted from multi type derivations, as shown by Accattoli et al. [ACSC21, AGL21].

In particular, we study CbV solvability via multi types. Our contributions are:

1. *Multi types and CbV solvability*: we characterize CbV solvability using Ehrhard's CbV *multi types* [Ehr12], which are strongly related to linear logic. Namely, we prove that a term is CbV solvable if and only if it is typable with a certain kind of multi types deemed *solvable* and inspired by Paolini and Ronchi Della Rocca [PRDR99];
2. *Bounds from types*: refining our solvable types, we extract the number of steps of solving reduction on a solvable term, together with the size of the solving normal form. This study re-casts de Carvalho's results in CbV, but it also requires new concepts.

While solvability is certainly subtler in CbV than in CbN, our contributions show that, if the presentation of CbV is carefully crafted, then a solid theory of CbV solvability is possible. In fact, we obtain a theory comparable to the one in CbN. This is our main achievement.

References

[ACSC21] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively. In *LICS*, pages 1–14. IEEE, 2021.

[AD12] Beniamino Accattoli and Ugo Dal Lago. On the Invariance of the Unitary Cost Model for Head Reduction. In *RTA*, pages 22–37, 2012.

[AG16] Beniamino Accattoli and Giulio Guerrieri. Open Call-by-Value. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016*, volume 10017 of *Lecture Notes in Computer Science*, pages 206–226. Springer, 2016.

[AGL21] Beniamino Accattoli, Giulio Guerrieri, and Maico Leberle. Semantic bounds and strong call-by-value normalization. *CoRR*, abs/2104.13979, 2021.

[AP12] Beniamino Accattoli and Luca Paolini. Call-by-value solvability, revisited. In *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, pages 4–16, 2012.

[Bar71] Hendrik Pieter Barendregt. *Some extensional term models for combinatory logics and l - calculi*. PhD thesis, Univ. Utrecht, 1971.

[Bar74] Hendrik Pieter Barendregt. Solvability in lambda-calculi. *Journal of Symbolic Logic - JSYML*, pages 372–372, 01 1974.

[CG14] Alberto Carraro and Giulio Guerrieri. A semantical and operational account of call-by-value solvability. In *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 103–118, 2014.

[dC07] Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. Thèse de doctorat, Université Aix-Marseille II, 2007.

[dC18] Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Math. Str. in Comput. Sci.*, 28(7):1169–1203, 2018.

[Ehr12] Thomas Ehrhard. Collapsing non-idempotent intersection types. In *CSL*, pages 259–273, 2012.

[EHRDR92] Lavinia Egidi, Furio Honsell, and Simona Ronchi Della Rocca. Operational, denotational and logical descriptions: a case study. *Fundam. Inform.*, 16(1):149–169, 1992.

[Gar94] Philippa Gardner. Discovering needed reductions using type theory. In *TACS '94*, volume 789 of *Lecture Notes in Computer Science*, pages 555–574. Springer, 1994.

[GN16] Álvaro García-Pérez and Pablo Nogueira. No solvable lambda-value term left behind. *Logical Methods in Computer Science*, 12(2), 2016.

[GPR17] Giulio Guerrieri, Luca Paolini, and Simona Ronchi Della Rocca. Standardization and conservativity of a refined call-by-value lambda-calculus. *Logical Methods in Computer Science*, 13(4), 2017.

[KMRDR21] Axel Kerinec, Giulio Manzonetto, and Simona Ronchi Della Rocca. Call-by-value, again! In *FSCD*, volume 195 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[MPRDR19] Giulio Manzonetto, Michele Pagani, and Simona Ronchi Della Rocca. New semantical insights into call-by-value λ-calculus. *Fundam. Inform.*, 170(1-3):241–265, 2019.

[Pao01] Luca Paolini. Call-by-value separability and computability. In *Theoretical Computer Science, 7th Italian Conference, ICTCS 2001, Torino, Italy, October 4-6, 2001, Proceedings*, pages 74–89, 2001.

[Plo75] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[PRDR99] Luca Paolini and Simona Ronchi Della Rocca. Call-by-value solvability. *RAIRO Theor. Informatics Appl.*, 33(6):507–534, 1999.

[PRR99] Alberto Pravato, Simona Ronchi Della Rocca, and Luca Roversi. The call-by-value λ-calculus: a semantic investigation. *Math. Str. in Comput. Sci.*, 9(5):617–650, 1999.

[RP04] Simona Ronchi Della Rocca and Luca Paolini. *The Parametric λ-Calculus – A Metamodel for Computation.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[Wad71] Christopher P. Wadsworth. *Semantics and pragmatics of the lambda-calculus.* PhD Thesis, Oxford, 1971. Chapter 4.

[Wad76] Christopher P. Wadsworth. The Relation Between Computational and Denotational Properties for Scott's D_∞-Models of the Lambda-Calculus. *SIAM J. Comput.*, 5(3):488–521, 1976.

An Agda Formalisation of Modalities and Erasure in a Dependently Typed Language

Oskar Eriksson and Andreas Abel

Department of Computer Science and Engineering, Chalmers and Gothenburg University,
{oskeri,abela}@chalmers.se

Modal types, like their counterparts in logic, allow modifiers to be attached to types. Given different sets of modifiers and different rules for the treatment of such modifiers by the type system, this allows types to be given a range of different interpretations. These interpretations can range from being quantitive in nature [6, 4], expressing, for instance, linearity or erasure [10, 12], to a variety of other interpretations such as data privacy [2]. Often, modal type systems are designed with specific interpretations in mind and the modifiers and rules are chosen based on this interpretation. An example of this is McBride's modality for erasure and linearity [9]. Others treat modal types in a more general sense and use a more general set of rules to allow different interpretations to be used in the same system.

There are different approaches to generalizing modalities [11, 8]. We have made a formalisation[1] in Agda of one such system for a dependently typed language [7], based on the ideas presented by Abel [1, 2] and Bernardy [2]. A modality is a ring-like structure whose elements are used to annotate terms in order to achieve the desired interpretation. By varying the structure, one can achieve a wide range of different interpretations but the common algebraic properties of the structures are used to define a type system that can check that annotations have been put down correctly.

The Agda formalisation (ca. 26000 lines of code) builds on a formalisation of decidability of type conversion by Abel et al. [3] (ca. 15000 lines of code) with the following novelties:

1. We adapt the syntax and typing judgements with modality annotations.
2. We introduce a new typing judgement for checking the validity of annotations.
3. As a case study, we instantiate it to the erasure modality with extraction to an untyped language.

Most closely related to our work is the Agda formalization by Wood [13] of a simply typed version of our calculus. In comparison to Atkey [5], our calculus also features a weak and a strong Σ-type, but we omit it in the following for lack of space.

The typing judgement for modality annotations relates modality contexts (assigning a modality element to each free variable) with terms. It is defined as follows, making use of the ring-like structure of the modality elements with its operations lifted to act pointwise on contexts γ, δ, a module for this ring.

$$\frac{}{\mathbf{0} \rhd \mathsf{U}} \qquad \frac{}{\mathbf{0} \rhd \mathbb{N}} \qquad \frac{\gamma \rhd F \quad \delta, q \rhd G}{\gamma + \delta \rhd \Pi^q_p \, F \, G} \qquad \frac{}{\mathbf{e}_i \rhd x_i} \qquad \frac{\gamma, p \rhd t}{\gamma \rhd \lambda^p \, t}$$

$$\frac{\gamma \rhd t \quad \delta \rhd u}{\gamma + p\delta \rhd t \, {}^p u} \qquad \frac{}{\mathbf{0} \rhd \mathsf{zero}} \qquad \frac{\gamma \rhd t}{\gamma \rhd \mathsf{suc}\, t} \qquad \frac{\gamma \rhd t}{\delta \rhd t} \, \delta \leq \gamma$$

[1] Available at https://fhlkfy.github.io/modalities_and_erasure/Logrel-MLTT.html

The erasure case study considers a modality with two elements, 0 and ω which are used to annotate computationally irrelevant and relevant terms respectively. For instance, $\lambda^0\, t$ represents a function whose argument is not used during computation whereas $\lambda^\omega\, t$ represents a function whose argument is. The point of using these erasure annotations is, of course, to achieve a kind of optimisation in which terms that have been marked as erasable can be removed. For this, we use an erasure function $\_^\bullet$ which translates terms into the untyped lambda calculus while removing erasable terms. Most notably $(t\,{}^0u)^\bullet = t\,\lightning$ where the erasable function argument u is replaced with $\lightning$, representing an undefined value.

If sound, applying this erasure function should not affect the result of fully evaluating a closed term. The proof of soundness makes use of two logical relations. The first, *reducibility* of terms is designed such that a term belongs to the relation iff it reduces to canonical form. For types, it is defined inductively as below. The Π-type case makes use of the reducibility relation for terms of some type F. This relation is defined recursively over $\mathscr{F}$, a proof that F is a reducible type.

$$\langle \mathsf{U} \rangle \frac{}{\Vdash_\ell \mathsf{U}_{\ell'}}\, \ell' < \ell \qquad \langle \mathbb{N} \rangle \frac{A \longrightarrow^* \mathbb{N}}{\Vdash_\ell A}$$

$$\langle \Pi \rangle \frac{A \longrightarrow^* \Pi^q_p\, F\, G \qquad \mathscr{F} : \Vdash_\ell F \qquad \forall a.\ (\Vdash_\ell a : F/\mathscr{F}) \Rightarrow (\Vdash_\ell G[a])}{\Vdash_\ell A}$$

The fundamental lemma for this relation is that $\vdash A$ and $\vdash t : A$ imply $\mathscr{A} : \Vdash_\ell A$ and $\Vdash_\ell t : A/\mathscr{A}$. This relation, in a more general form not restricted to closed terms, was part already in the formalisation by Abel et al. [3].

The second relation relates closed terms in the source language with closed terms in the target language, $t \mathbin{®_\ell} v : A/\mathscr{A}$ and is also defined by recursion on $\mathscr{A}$, a proof that A is reducible:

- If $\mathscr{A} = \langle \mathsf{U} \rangle$ then $t \mathbin{®_\ell} v : A/\mathscr{A}$ holds iff $\vdash t : \mathsf{U}$.
- If $\mathscr{A} = \langle \mathbb{N} \rangle$ then $t \mathbin{®_\ell} v : A/\mathscr{A}$ holds iff either
  - $t \longrightarrow^* \mathsf{zero} : \mathbb{N}$ and $v \longrightarrow^* \mathsf{zero}$, or
  - $t \longrightarrow^* \mathsf{suc}\, t' : \mathbb{N}$ and $v \longrightarrow^* \mathsf{suc}\, v'$ and $t' \mathbin{®_\ell} v' : A/\mathscr{A}$.
- If $\mathscr{A} = \langle \Pi \rangle$, then $A \longrightarrow^* \Pi^q_p\, F\, G$ holds and there are derivations $\mathscr{F} : \Vdash_\ell F$ and $\mathscr{G} : \forall a.\ \Vdash_\ell a : F/\mathscr{F} \Rightarrow \Vdash_\ell G[a]$. We then define $t \mathbin{®_\ell} v : A/\mathscr{A}$ to hold iff either
  - $p = 0$ and $t\,{}^0a \mathbin{®_\ell} v\,\lightning : G[a]/\mathscr{G}(a)$ for all $\Vdash_\ell a : F/\mathscr{F}$, or
  - $p = \omega$ and $t\,{}^\omega a \mathbin{®_\ell} v\, w : G[a]/\mathscr{G}(a)$ for all $\Vdash_\ell a : F/\mathscr{F}$ and $a \mathbin{®_\ell} w : F/\mathscr{F}$.

Especially noteworthy is the case for Π-types where, non-erased function terms, t and v are related if they are related when applied to related arguments. For erasable functions, however, the argument on the target language side is replaced with $\lightning$, mirroring the definition of the extraction function.

The fundamental lemma for this relation is that $\vdash t : A$ and $\triangleright\, t$ imply $t \mathbin{®_\ell} t^\bullet : A/\mathscr{A}$. Using this property, the extraction function can be shown to be sound. In particular, all terms of type $\mathbb{N}$ represent the same natural number before and after extraction.

References

[1] Andreas Abel. Resourceful dependent types. In *Presentation at 24th International Conference on Types for Proofs and Programs (TYPES 2018), Braga, Portugal*, 2018. abstract.

[2] Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proc. ACM Program. Lang.*, 4(ICFP):90:1–90:28, 2020.

[3] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[4] Robert Atkey. The syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 9-12, 2018*, pages 56–65. ACM Press, 2018.

[5] Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018.

[6] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *Programming Languages and Systems. ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014.

[7] Oskar Eriksson. An Agda formalization of modalities and erasures in a dependently typed language. Master's thesis, Chalmers University of Technology, 2021.

[8] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. *Log. Methods Comput. Sci.*, 17(3), 2021.

[9] Conor McBride. I got plenty o'nuttin'. In *A List of Successes That Can Change the World*, pages 207–233. Springer, 2016.

[10] Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In *International Conference on Foundations of Software Science and Computational Structures*, pages 350–364. Springer, 2008.

[11] Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. *J. Log. Algebraic Methods Program.*, 120:100637, 2021.

[12] Matúš Tejiščák. A dependently typed calculus with pattern matching and erasure inference. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[13] James Wood and Robert Atkey. A linear algebra approach to linear metatheory. 353:195–212, 2020.

Aspects of a machine-checked intermediate language for extraction from Coq, in MetaCoq

Yannick Forster and Matthieu Sozeau

Inria, Gallinette Project-Team, Nantes, France

Abstract

We discuss our progress on establishing $\lambda_\Box$, an untyped version of the PCUIC calculus used in MetaCoq to model the type theory of the Coq proof assistant, as intermediate language for various verified extraction and compilation projects. We aim at establishing and unifying correctness properties for erasure from PCUIC to $\lambda_\Box$ and at providing both translation-validated and verified syntax transformations on PCUIC and $\lambda_\Box$.

In previous work, we have given a machine-checked correctness proof for type and proof erasure [10] based on MetaCoq [9]. The central theorem is due to Letouzey [8] and states that if a term t of PCUIC (the name of the calculus used in the MetaCoq formalisation of the type theory underlying Coq) weak call-by-value evaluates to a value v, then the application of the erasure function to t yields a term in the untyped calculus $\lambda_\Box$ which weak call-by-value evaluates to a value v' which is in the erasure relation of v. Crucially, the theorem mentions both an erasure function and a (non-functional) erasure relation, which only agree on values of first-order inductive type but not in general (see [8, 10] for details).

Currently, $\lambda_\Box$ is used as intermediate language for extraction from Coq to other targets, e.g. in the CertiCoq compiler from Coq to C code [1, 3], in the ConCert project [2] extracting from Coq to (ML-like) blockchain languages, and in our own project on verifying Coq's extraction to OCaml.[1] In the latter, we are working with respect to a formal semantics of the Malfunction language, the (untyped) intermediate language of the OCaml compiler [5].

The challenges arising are manifold: They are of mathematical nature, often due to the intricacy of PCUIC regarding universes and cases, proof-engineering challenges, often due to the large size of terms (with 15 constructors) and predicates (the cumulativity relation has 26 rules), and software engineering challenges, usually due to the > 20 minutes build time of the project, consisting of more than 150k LoC.

First-order inductive types and values We define a boolean function inspecting the syntactic representation of an inductive type and checking whether it is first-order. An inductive type is first-order if the types of all parameters, indices, and arguments of constructors are first-order. With this definition `nat` is first-order, `list` and `Vector.t` are not, but e.g. `listbool` and `vectorbool : nat -> Type` (non-polymorphic variants of `list bool` and `Vector.t bool`) are first-order. We prove that values of first-order inductive type can be characterised by an inductive predicate just admitting constructor applications of first-order inductive types, and that on values of first-order inductive types the erasure function and the erasure predicate agree.

Weak call-by-value evaluation vs. reduction In PCUIC, as usual in type theory, the reduction strategy is neither call-by-value nor call-by-name, and crucially applies to open terms.

The machine-checked correctness theorem of type and proof erasure, tailored to serve as the front end of extraction pipelines, states correctness w.r.t. weak call-by-value evaluation. It is easy to prove that weak call-by-value evaluation is included in reduction, but the converse is of course not true in general. It is however true for terms of inductive first-order type. We are working towards a machine-checked proof of this standardisation result, relying on a newly machine-checked proof of progress for weak call-by-value reduction and, for simplicity, on strong normalisation of reduction.

[1] Joint work with Pierre Giraud, Pierre-Marie Pédrot, and Nicolas Tabareau.

Constructors as functions vs. constructors as blocks In PCUIC, constructors are (curried) functions, e.g. `cons S` is a valid term. Both the CertiCoq representation of inductive types and constructor types in Malfunction are modelled after the OCaml representation, where constructors are n-ary but not functions (see [7] for historic comments).

Currently, PCUIC does not model η-equivalence, as explained in [6]. We thus define an η-expandedness predicate for both PCUIC and $\lambda_\Box$, and implement a syntax transformation returning η-expanded terms. The Coq kernel can be used to certify that the result is indeed (η-)convertible to the original term, a translation validation approach first used in ConCert [2].

Structural fixpoints vs. true fixpoints In PCUIC, recursion is structural and fixpoints are annotated with a principal argument. Fixpoint reduction is triggered once the principal argument exposes a constructor application. We mirror this behaviour also in $\lambda_\Box$ to simplify the correctness proof of erasure. In particular, this means that fixpoints always are functions, and fixpoints applied to some but not enough arguments are considered values.

Programming languages with non-termination do not require structural recursion. In the first intermediate language L2k of the CertiCoq compiler, a fixpoint reduces once a single argument is present, and only non-applied fixpoints are considered values. In Malfunction, the body of a fixpoint needs to be syntactically a λ-expression, and a fixpoint is always unfolded, consequently fixpoints are not considered values.

In general, translating from structural fixpoints in $\lambda_\Box$ to unary fixpoints (treated either like in Malfunction or like in L2k) will not be correct. We identify a subset of PCUIC for which the translation behaves correctly: Fixpoints have to be always syntactically applied to at least $1 + r$ arguments, where r is the index of the structural recursion argument, and the variable corresponding to a recursive call in the body of a fixpoint has to be expanded as well.

We provide a second weak call-by-value evaluation relation for $\lambda_\Box$ which is equivalent on terms with expanded fixpoints. The η-expansion translation also covering fixpoints becomes significantly more involved than our previous version and the one used in ConCert.

Case representation PCUIC represents cases with explicit contexts, containing also the bodies bound in `let` expressions in constructor types [11]. $\lambda_\Box$ does not allow `let` expressions in case contexts. Thus, we provide a verified `let`-expansion pass on PCUIC, where `let`s in constructor types are disallowed, simplifying other future syntax transformations on PCUIC.

Parameter stripping Parameters are irrelevant for case analysis in PCUIC, and are not represented in terms in polymorphically typed languages like OCaml. Thus, it makes sense to remove parameters from inductive types before extracting from $\lambda_\Box$ to a programming language. We provide such a machine-checked parameter stripping pass. We prove that it does not affect weak call-by-value evaluation on eta-expanded terms, and that it preserves eta-expansiveness. Although mathematically not involved, the verification requires around 1500 lines of proofs.

Induction principles and views Application in PCUIC is unary, but e.g. mutual fixpoints contain a list of their bodies. The former makes inspecting the head of an application in recursive functions tedious, while for the latter Coq's generated induction principles do not suffice. We provide views usable with the Equations plugin [12] and manually proved induction lemmas to solve these issues.

Performance Our erasure process can both be used by extracting it to OCaml and inside Coq. We are working on improving performance for both. Building the global context only once, maintaining it through all translations, and using efficient update mechanisms via AVL trees turned out to be crucial and yielded performance gains with a factor of 100. We are currently investigating whether using extensional tries [4] yields further performance gains.

References

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[2] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: a smart contract certification framework in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 215–228. ACM, 2020. doi:10.1145/3372885.3373829.

[3] Andrew Appel, Yannick Forster, Anvay Grover, Joomy Korkut, John Li, Zoe Paraskevopoulou, and Matthieu Sozeau. CertiCoq (GitHub repository). 2022. URL: https://github.com/CertiCoq/certicoq.

[4] Andrew W Appel and Xavier Leroy. Efficient Extensional Binary Tries. working paper or preprint, October 2021. URL: https://hal.inria.fr/hal-03372247.

[5] Stephen Dolan. Malfunctional programming. In *ML Workshop*, 2016.

[6] Meven Lennon-Bertrand. À bas l'η — Coq's troublesome η-conversion. In *The first Workshop on the Implementation of Type Systems (WITS)*, 2022.

[7] Xavier Leroy. Post on the Caml mailing list. 2001. URL: https://web.archive.org/web/20170822231903/https://caml.inria.fr/pub/ml-archives/caml-list/2001/08/47db53a4b42529708647c9e81183598b.en.html.

[8] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. PhD thesis, 2004. URL: http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf.

[9] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 2020. URL: https://hal.inria.fr/hal-02167423, doi:10.1007/s10817-019-09540-0.

[10] 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):1–28, 2019. doi:10.1145/3371076.

[11] Matthieu Sozeau, Meven Lennon-Bertrand, and Yannick Forster. The curious case of case: correct & efficient representation of case analysis in coq and metacoq. In *The first Workshop on the Implementation of Type Systems (WITS)*, 2022.

[12] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019. doi:10.1145/3341690.

BioTT: a Family of Brouwerian Intuitionistic Theories Open to Classical Reasoning

Mark Bickford[1], Liron Cohen[2], Robert L. Constable[1], and Vincent Rahli[3]

[1] Cornell University, USA [2] Ben-Gurion University, Israel [3] University of Birmingham, UK

1 Introduction

One difference between Brouwerian intuitionistic logic and classical logic is their treatment of time. In classical logic truth is atemporal, whereas in intuitionistic logic it is time-relative. An example of a time-relative notion is that of Brouwer's choice sequences, which are finite sequences of entities (e.g., natural numbers) that are never complete, and can always be further extended with new choices [16; 5; 25; 26; 19; 27; 22]. This manifestation of the evolving concept of time in intuitionistic logic entails a notion of computability that goes far beyond that of Church-Turing [11, Sec.5]. Brouwer used this concept among other things to define the continuum [4, Ch.3], and it has further been used in so-called intuitionistic (weak) counterexamples to derive the negation of classical axioms such as the Law of Excluded Middle (LEM) [15; 9; 20; 17].

We have built an intuitionistic extensional type theory called BITT [7] (see https://github.com/vrahli/NuprlInCoq/tree/beth for its Coq formalization), which features choice sequences and is given meaning through a Beth model [28; 6; 14, Sec.145; 12, Sec.5.4; 11]. We have showed that this theory is anti-classical following the counterexamples mentioned above. This is not only due to the presence of choice sequences, but also to its Beth model, which provides an anti-classical notion of time, which forces some properties on choice sequences to be undecidable. We subsequently developed another intuitionistic extensional type theory called OpenTT [8] (see https://github.com/vrahli/NuprlInCoq/tree/ls3/ for its Coq formalization), which is given meaning through a "relaxed" notion of a Beth model, called the *open bar* model, which sufficiently weakens the "undecided" nature of choice sequences to enable validating classical axioms, such as LEM. OpenTT also features choice sequences, and includes standard choice sequence axioms, namely: the Axiom of Open Data, a density axiom, and a discreteness axiom [23; 24; 18; 11].

These two theories and models share a common core, which forms the basis for a family of extensional type theories with choice sequences, that can either be made anti-classical or classical-compatible depending on the chosen model. This family provides a computational setting for exploring the implications of time-relative constructs such as choice sequences, For example, it can enable the development of constructive Brouwerian real number theories. It also provides a mean to capture a more relaxed notion of time, providing a basis for more classically-inclined Brouwerian intuitionistic theories. Let us now describe this family of theories at a high level, which we call BioTT here for *Brouwerian Intuitionistic Open Type Theories*.

2 World-Based Calculus

BioTT relies on a untyped call-by-name λ-calculus, whose core syntax includes:

$$
\begin{array}{l}
v \in \texttt{Value} ::= vt \mid \star \mid \underline{n} \mid v \mid \lambda x.t \mid \texttt{inl}(t) \mid \texttt{inr}(t) \mid \langle t_1, t_2 \rangle \\
vt \in \texttt{Type} ::= \mathbb{N} \mid t_1 < t_2 \mid \mathbb{U}_i \mid \mathbf{\Pi} x{:}t_1.t_2 \mid \mathbf{\Sigma} x{:}t_1.t_2 \mid \{x : t_1 \mid t_2\} \mid t_1 + t_2 \mid t_1 = t_2 \in t \mid \texttt{Free} \\
t \in \texttt{Term} ::= x \mid v \mid t_1\ t_2 \mid \texttt{let}\ x, y = t_1\ \texttt{in}\ t_2 \mid \texttt{fix}(t) \mid \texttt{case}\ t\ \texttt{of}\ \texttt{inl}(x) \Rightarrow t_1 \mid \texttt{inr}(y) \Rightarrow t_2
\end{array}
$$

with numbers $\underline{n}$ as primitives, injections, pairs, where x is a variable, and where v is a choice sequence name, which inhabit the type `Free` of *free* choice sequences (see [7; 8] for further details). In BioTT, a choice sequence is implemented as a *name*, which allows referring to the

sequence in computations (see below), along with its current state, which is a list of choices, e.g., a list of numbers for a choice sequence of numbers.

BioTT's core small-step call-by-name operational semantics allows in particular (1) β-reducing applications of λ-abstractions; (2) unrolling fixpoints, (3) examining choice sequences; (4) destructing pairs; and (5-6) destructing injections:

(1) $(\lambda x.t)\ u \mapsto_w t[x\backslash u]$
(2) $\mathtt{fix}(v) \mapsto_w v\ \mathtt{fix}(v)$
(3) $v(\underline{n}) \mapsto_w w[v][\underline{n}]$
(4) $\mathtt{let}\ x, y = \langle t_1, t_2\rangle\ \mathtt{in}\ t \mapsto_w t[x\backslash t_1; y\backslash t_2]$
(5) $\mathtt{case\ inl}(t)\ \mathtt{of\ inl}(x) \Rightarrow t_1\ \mid\ \mathtt{inr}(y) \Rightarrow t_2 \mapsto_w t_1[x\backslash t]$
(6) $\mathtt{case\ inr}(t)\ \mathtt{of\ inl}(x) \Rightarrow t_1\ \mid\ \mathtt{inr}(y) \Rightarrow t_2 \mapsto_w t_2[y\backslash t]$

Note that this semantics is parameterized by a *world* w. BioTT is parameterized by a Kripke frame [21; 20] consisting of a set of *worlds* $\mathcal{W}$ equipped with a reflexive and transitive binary relation $\sqsubseteq$. To support computing with choice sequences, worlds allow storing choice sequences, and the above semantics allows applying the name of a choice sequence v to a number $\underline{n}$ in order to access the $\underline{n}^{th}$ choice made for v in the current world w, written $w[v][\underline{n}]$.

BioTT's core inference rules include standard sequent calculus rules such as the following $\mathbf{\Pi}$-introduction rule, where H is a list of hypotheses (see [7; 8] for details): if $H, z : A \vdash b : B[x\backslash z]$ and $H \vdash A \in \mathbb{U}_i$ then $H \vdash \lambda z.b : \mathbf{\Pi} x{:}A.B$.

3 Bar-Based Forcing Semantics

BioTT is given meaning through a family of forcing interpretations, where types are interpreted as Partial Equivalence Relations [1; 2; 10; 3], which are parameterized by a bar notion B. A bar b is a set of world extending (w.r.t. $\sqsubseteq$) a given world w (we write b_w to indicate the world w that b *bars*), and a bar "notion" is then a predicate B on bars. This interpretation is inductively-recursively [13] defined as (1) an inductive relation $w \vDash T_1 \mathrel{\equiv} T_2$ that expresses type equality; and (2) a recursive function $w \vDash t_1 \mathrel{\equiv} t_2 \mathrel{\in} T$ that expresses equality in a type. In particular, this interpretation is closed under bars as follows:

$$w \vDash T_1 \mathrel{\equiv} T_2 \iff \exists b_w.\forall w' \in b_w.\exists T_1', T_2'.(T_1 \Downarrow_{w'} T_1' \wedge T_2 \Downarrow_{w'} T_2' \wedge w' \vDash T_1' \mathrel{\equiv} T_2')$$
$$w \vDash t_1 \mathrel{\equiv} t_2 \mathrel{\in} T \iff \exists b_w.\forall w' \in b_w.\exists T'.(T \Downarrow_{w'} T' \wedge w' \vDash t_1 \mathrel{\equiv} t_2 \mathrel{\in} T')$$

where $T \Downarrow_w T'$ states that T computes to T' in all extensions (w.r.t. $\sqsubseteq$) of w.

We can then show that according to this interpretation, BioTT forms a type system, in the sense that $w \vDash T_1 \mathrel{\equiv} T_2$ and $w \vDash t_1 \mathrel{\equiv} t_2 \mathrel{\in} T$ are symmetric, transitive, and respect computation, and are also monotonic and local as expected for such possible-world semantics [28; 14; 12, Sec.5.4]. In particular, this is true when the predicate B specifies a topological space of bars.

Beth Bars. B can be instantiated so as to capture Beth bars as follows. A bar b of a world w is a Beth bar iff for all infinite chains of extensions $w \sqsubseteq w_1 \sqsubseteq w_2 \sqsubseteq \ldots$, there exists an $i \in \mathbb{N}$ such that $w_i \in b$. The resulting model allows validating the following axioms [7]:

- density: $\mathbf{\Pi} n{:}\mathbb{N}.\mathbf{\Pi} f{:}\mathcal{B}_n.{\downarrow}\mathbf{\Sigma} a{:}\mathtt{Free}.f = a \in \mathcal{B}_n$
- discreteness: $\mathbf{\Pi} a, b{:}\mathtt{Free}.(a = b \in \mathcal{B}) + \neg(a = b \in \mathcal{B})$
- $\neg$LEM: $\neg\mathbf{\Pi} P{:}\mathbb{U}_i.{\downarrow}(P + \neg P)$

where $\mathbb{N}_n \coloneqq \{k : \mathbb{N} \mid k < n\}$; $\mathcal{B} \coloneqq \mathbb{N} \to \mathbb{N}$; $\mathcal{B}_n \coloneqq \mathbb{N}_n \to \mathbb{N}$; $\mathtt{True} \coloneqq \underline{0} = \underline{0} \in \mathbb{N}$; $\mathtt{False} \coloneqq \underline{0} = \underline{1} \in \mathbb{N}$; $\neg T \coloneqq T \to \mathtt{False}$; and ${\downarrow}T \coloneqq \{x : \mathtt{True} \mid T\}$.

Open Bars. B can be instantiated so as to capture open bars as follows. A bar b of a world w is an open bar iff for all $w_1 \sqsupseteq w$, there exists a $w_2 \sqsupseteq w_1$ such that for all $w_3 \sqsupseteq w_2$, $w_3 \in b$. The resulting model allows validating the following axioms [8]:

- open data: $\mathbf{\Pi}\alpha{:}\mathtt{Free}.P(\alpha) \to {\downarrow}\mathbf{\Sigma} n{:}\mathbb{N}.\mathbf{\Pi}\beta{:}\mathtt{Free}.(\alpha = \beta \in \mathcal{B}_n \to {\downarrow}P(\beta))$
- density: $\mathbf{\Pi} n{:}\mathbb{N}.\mathbf{\Pi} f{:}\mathcal{B}_n.{\downarrow}\mathbf{\Sigma} a{:}\mathtt{Free}.f = a \in \mathcal{B}_n$
- discreteness: $\mathbf{\Pi} a, b{:}\mathtt{Free}.(a = b \in \mathcal{B}) + \neg(a = b \in \mathcal{B})$
- LEM: $\mathbf{\Pi} P{:}\mathbb{U}_i.{\downarrow}(P + \neg P)$

References

[1] Stuart F. Allen. "A Non-Type-Theoretic Definition of Martin-Löf's Types". In: *LICS*. IEEE Computer Society, 1987, pp. 215–221.

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

[3] Abhishek Anand and Vincent Rahli. "Towards a Formally Verified Proof Assistant". In: *ITP 2014*. Vol. 8558. LNCS. Springer, 2014, pp. 27–44. DOI: 10.1007/978-3-319-08970-6_3.

[4] Mark van Atten. *On Brouwer*. Wadsworth Philosophers. Cengage Learning, 2004.

[5] Mark van Atten and Dirk van Dalen. "Arguments for the continuity principle". In: *Bulletin of Symbolic Logic* 8.3 (2002), pp. 329–347.

[6] Evert Willem Beth. *The foundations of mathematics: A study in the philosophy of science*. Harper and Row, 1966.

[7] Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. "Computability Beyond Church-Turing via Choice Sequences". In: *LICS 2018*. ACM, 2018, pp. 245–254. DOI: 10.1145/3209108.3209200.

[8] Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. "Open Bar - a Brouwerian Intuitionistic Logic with a Pinch of Excluded Middle". In: *CSL*. Vol. 183. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 11:1–11:23. DOI: 10.4230/LIPIcs.CSL.2021.11.

[9] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. London Mathematical Society Lecture Notes Series. Cambridge University Press, 1987.

[10] Karl Crary. "Type-Theoretic Methodology for Practical Programming Languages". PhD thesis. Ithaca, NY: Cornell University, Aug. 1998.

[11] Dirk van Dalen. "An interpretation of intuitionistic analysis". In: *Annals of mathematical logic* 13.1 (1978), pp. 1–43.

[12] Michael A. E. Dummett. *Elements of Intuitionism*. Second. Clarendon Press, 2000.

[13] Peter Dybjer. "A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory". In: *J. Symb. Log.* 65.2 (2000), pp. 525–549.

[14] VH Dyson and Georg Kreisel. *Analysis of Beth's semantic construction of intuitionistic logic*. Stanford University. Applied Mathematics and Statistics Laboratories, 1961.

[15] Arend Heyting. *Intuitionism: an introduction*. North-Holland Pub. Co., 1956.

[16] Stephen C. Kleene and Richard E. Vesley. *The Foundations of Intuitionistic Mathematics, especially in relation to recursive functions*. North-Holland Publishing Company, 1965.

[17] Georg Kreisel. "A Remark on Free Choice Sequences and the Topological Completeness Proofs". In: *J. Symb. Log.* 23.4 (1958), pp. 369–388. DOI: 10.2307/2964012.

[18] Georg Kreisel. "Lawless sequences of natural numbers". In: *Compositio Mathematica* 20 (1968), pp. 222–248.

[19] Georg Kreisel and Anne S. Troelstra. "Formal systems for some branches of intuitionistic analysis". In: *Annals of Mathematical Logic* 1.3 (1970), pp. 229–387. DOI: http://dx.doi.org/10.1016/0003-4843(70)90001-X.

[20] Saul A. Kripke. "Semantical Analysis of Intuitionistic Logic I". In: *Formal Systems and Recursive Functions*. Vol. 40. Studies in Logic and the Foundations of Mathematics. Elsevier, 1965, pp. 92–130. DOI: https://doi.org/10.1016/S0049-237X(08)71685-9.

[21] Saul A. Kripke. "Semantical Analysis of Modal Logic I. Normal Propositional Calculi". In: *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik* 9.5-6 (1963), pp. 67–96. DOI: 10.1002/malq.19630090502.

[22] Joan R. Moschovakis. "An intuitionistic theory of lawlike, choice and lawless sequences". In: *Logic Colloquium'90: ASL Summer Meeting in Helsinki*. Association for Symbolic Logic. 1993, pp. 191–209.

[23] Joan Rand Moschovakis. *Choice Sequences and Their Uses*. 2015.

[24] A. S. Troelstra. "Analysing choice sequences". In: *J. Philosophical Logic* 12.2 (1983), pp. 197–260. DOI: 10.1007/BF00247189.

[25] Anne S. Troelstra. "Choice Sequences and Informal Rigour". In: *Synthese* 62.2 (1985), pp. 217–227.

[26] Anne S. Troelstra. *Choice sequences: a chapter of intuitionistic mathematics*. Clarendon Press Oxford, 1977.

[27] Wim Veldman. "Understanding and Using Brouwer's Continuity Principle". In: *Reuniting the Antipodes — Constructive and Nonstandard Views of the Continuum*. Vol. 306. Synthese Library. Springer Netherlands, 2001, pp. 285–302. DOI: 10.1007/978-94-015-9757-9_24.

[28] Beth E. W. "Semantic Construction of Intuitionistic Logic". In: *Journal of Symbolic Logic* 22.4 (1957), pp. 363–365.

Canonicity and decidability of equality for setoid type theory

István Donkó[1]*and Ambrus Kaposi[1]†

Eötvös Loránd University, Budapest, Hungary
{isti115,akaposi}@inf.elte.hu

Abstract

A proof assistant based on Setoid Type Theory would be practical for dealing with quotient inductive types. In order for such a system to be implemented in a proof assistant, we have to prove that it has decidability of equality. Our aim is to verify this property by constructing a syntactic translation from the syntax of Martin-Löf Type Theory and showing that this translation is injective.

1 Motivation

Type theory has proven to be an indispensable tool for precisely formalizing statements as well as constructing and validating proofs for them. In its current implementations handling some problems is quite cumbersome though. For example dealing with quotient types involves either lots of extra manual work, because the added equalities need to be eliminated manually (so called "transport hell") or the other option is to enable certain language features (e.g. rewrite rules in Agda [**?**]) which in turn deteriorate the computational properties of the type theory.

One possible alleviation of this problem could be basing proof assistants on Setoid Type Theory (SeTT, [**?**, **?**]), also called observational type theory ([**?**, **?**]), which would natively enable the usage of quotient inductive types. SeTT is based on the setoid model where the equality relation for any type can be specified arbitrarily.

2 Requirements

For SeTT to be usable for such purposes, it needs to have certain properties such as canonicity and decidable equality which is necessary for type checking. Instead of proving these properties using usual methods such as logical relations [**?**], we simplify the procedure using a model construction. We call the model construction *setoidification*. We use the variant of the model construction described in [**?**]. Any model of Martin-Löf type theory with strict propositions (MLTTP) can be turned into a model of SeTT. We aim to transport the necessary properties of the model of MLTTP to its setoidified version. In particular, we want to prove that the interpretation of the syntax of SeTT into the setoidified syntax of MLTTP is injective. Injectivity can also be called completeness: every equality that holds in the setoidification of MLTTP is reflected in the syntax of SeTT.

*Supported by the ÚNKP-21-3 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

†Ambrus Kaposi was supported by the "Application Domain Specific Highly Reliable IT Solutions" project which has been implemented with support from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme, and by Bolyai Scholarship BO/00659/19/3.

A term in the setoidified model t is a term in the original model $|t|$ together with a proof that it respect the equivalence relation belonging to the type of the term. We call the projection $|\_|$ the *evaluation* function.

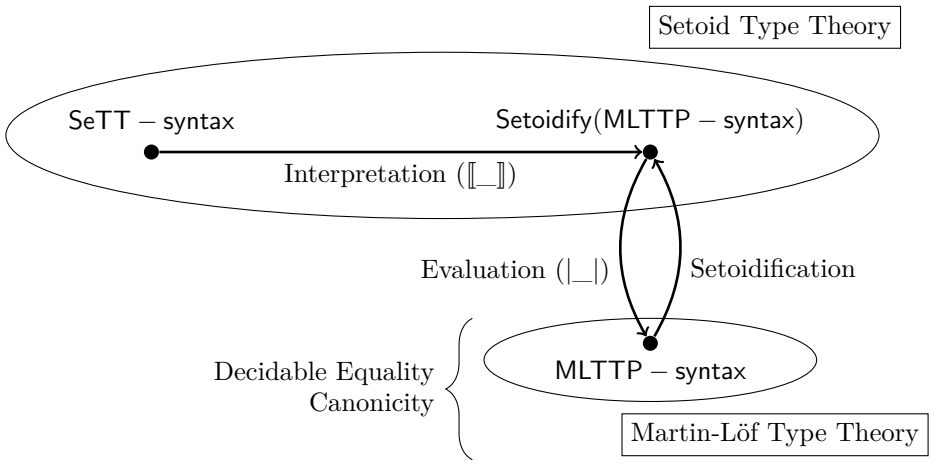


Figure 1: Setoidification illustrated

Let the $[\![\_]\!]$ operation be the interpretation from the syntax of SeTT to the setoidification of the syntax of MLTTP. We assume injectivity of the interpretation function, that is, for any t, t' terms in SeTT of the same type, $[\![t]\!] = [\![t']\!] \Rightarrow t = t'$.

- *Decidability of equality*
  Decidability of equality states that either $t = t'$ or $t \neq t'$ holds. Through interpretation and evaluation, $|[\![t]\!]| = |[\![t']\!]|$ is an equality in the base model. If we have decidable equality there, we have two cases:
  - $|[\![t]\!] = |[\![t']\!]|$ implies $[\![t]\!] = [\![t']\!]$ which implies $t = t'$ by injectivity
  - $|[\![t]\!]| \neq |[\![t']\!]|$ implies $t \neq t'$ by contradiction
- *Canonicity*
  Canonicity means that every closed term can be equated to one that is only built using the constructors of the given type. For example, the type Bool has two canonical forms in the base model:
  - $|[\![t]\!]| = \mathsf{false}_{\mathsf{MLTTP-syntax}}$ implies $t = \mathsf{false}_{\mathsf{SeTT-syntax}}$ by injectivity
  - $|[\![t]\!]| = \mathsf{true}_{\mathsf{MLTTP-syntax}}$ implies $t = \mathsf{true}_{\mathsf{SeTT-syntax}}$ by injectivity

We are in the process of proving injectivity of interpretation into setoidification following the analogous proof of injectivity of interpretation into termification [**?**].

References

[1] Agda Development Team. Agda - https://github.com/agda/agda.

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

[5] Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory is morally correct. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 329–365. Springer, 2019.

[6] Loïc Pujet and Nicolas Tabareau. Observational equality: now for good. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022.

Capable GV: Capabilities for Session Types in GV*

Magdalena J. Latifa[1] and Ornela Dardha[1]

[1] University of Glasgow
2398248L@student.gla.ac.uk
[2] University of Glasgow
ornela.dardha@glasgow.ac.uk

Abstract

This is an ongoing work which introduces Capable GV (CGV)–a functional calculus with binary session types that uses *capabilities* and allows channel sharing. Capabilities allow to split channels into two entities: a linear capability and an unrestricted channel endpoint. As channels themselves are unrestricted, this allows sharing, and thus increases expressivity wrt previous work. Orthogonality, CGV allows cyclic processes at the cost of losing deadlock freedom. Our aim is to restore deadlock freedom in the future by using priorities, as in works by Kokke and Dardha [6, 5]. This work presents the language terms, types and typing rules. Semantics and safety results are work in progress.

1 Introduction

Good Variation (GV) is a functional calculus with binary session types that allows protocol-compliant concurrent communication. It was originally introduced by Wadler [9], based on [4]. Since then, many extensions to GV have been developed, including Exceptional GV [3], Priority GV [6] and Hypersequent GV [2], with the aim of improving its expressivity and guarantees, while taking advantage of the benefits that GV provides: higher-order functions, separation of run-time configuration and a more natural fit for implementations. *Capabilities* [8] are a method of allowing channel sharing. This method relies on splitting channels into two disjoint components, which allows channels to be safely shared. In this work, we develop an extension to GV that allows channel sharing, consequently increasing the expressivity of the system.

2 Capable GV

We present the statics of Capable GV (CGV), a GV-based functional language with shareable session types. Channel sharing in CGV is achieved by introducing unrestricted channel endpoints and linear capabilities that are managed by a flow-sensitive type-and-effect system. As linearity is enforced via capabilities, sharing does not violate communication safety. While CGV allows for greater expressivity, it comes at the cost of deadlock-freedom as the system allows cyclic processes. CGV types are defined by the following grammar:

$$\begin{array}{rcl} S & ::= & !T.S \mid ?T.S \mid \mathbf{end} \\ T, U & ::= & T \times U \mid T + U \mid \mathbf{1} \mid tr(\rho) \mid [\rho(S)] \mid T\ (C_1) \multimap (C_2)\ U \\ \Gamma, \Delta & ::= & \varnothing \mid \Gamma, x : T \\ C & ::= & \varnothing \mid C \otimes \rho(S) \end{array}$$

Constructs $tr(\rho)$ and $\rho(S)$ are the core of CGV and represent the separation of the channel endpoint and the capability of using it. Tracked type $tr(\rho)$ specifies that the channel endpoint

*Work supported by the EU H2020 MSCA RISE project BehAPI ID 778233.

is controlled by capability ρ. Capability ρ exists in a capability set C in the form $\rho(S)$ which specifies the channel's session type S. Additionally, a capability can be packed into a pack type $[\rho(S)]$ and subsequently relayed to another thread. In order to accommodate for capabilities in the type-and-effect system, functions have the type $T\ (C_1) \multimap (C_2)\ U$ which denotes a linear function $T \multimap U$ where the function body requires pre-evaluation capability set C_1 and produces capability set C_2 when evaluated. The remaining types are standard session types or linear λ-calculus types.

CGV terms are defined by the following grammar:

$$
\begin{array}{rcl}
V, W & ::= & () \mid x \mid \lambda x.M \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V \\
L, M, N & ::= & V\ W \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M \mid \mathbf{let}\ () = V\ \mathbf{in}\ M \\
& \mid & \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \mid \mathbf{return}\ V \\
& \mid & \mathbf{new} \mid \mathbf{send}\ V \mid \mathbf{recv}\ V \mid \mathbf{close}\ V \mid \mathbf{pack}\ V \mid \mathbf{unpack}\ V \mid \mathbf{spawn}\ M\ N
\end{array}
$$

Terms $\mathbf{pack}\ V$ and $\mathbf{unpack}\ V$ are unique to CGV and allow "inactivating" and "reactivating" channels by packing and unpacking capabilities. Terms $\mathbf{send}\ V$ and $\mathbf{recv}\ V$ are standard to GV but they no longer return a copy of the channel since channels are unrestricted. Terms $\mathbf{close}\ V$, $\mathbf{new}$ and $\mathbf{spawn}\ M\ N$ are analogous to Priority GV's terms [6] with the change of $\mathbf{new}$ also creating capabilities for the new channels and $\mathbf{spawn}$ requiring a packed capability to initialise the newly spawned thread with. The rest of the terms are standard linear λ-calculus terms.

Typing rules T-Recv and T-Pack demonstrate the behaviour of capabilities in CGV. For T-Recv, in order to receive a message of type T on a channel of type $tr(\rho)$, the capability of using the endpoint $\rho(?T.S)$ needs to be in the pre-evaluation capability set. After this communication takes place, the capability must update the session type; hence the post-evaluation capability set must contain the capability in the form $\rho(S)$. T-Pack specifies the behaviour of packing the capability in order to relay it to another thread. In order to pack the capability of channel of type $tr(\rho)$, the capability $\rho(S)$ needs to be in the pre-evaluation capability set. After the channel is packed, the capability $\rho(S)$ is removed from the capability set and is instead "inactivated" and expressed in the pack type $[\rho(S)]$. The session type of this channel cannot change until the capability is unpacked and the channel becomes active again, which is key to ensuring linearity of communication.

$$
\text{T-Recv}\ \frac{\Gamma \vdash V : tr(\rho)}{\Gamma; C \otimes \rho(?T.S) \vdash \mathbf{recv}\ V : T \triangleright C \otimes \rho(S)} \qquad \text{T-Pack}\ \frac{\Gamma \vdash V : tr(\rho)}{\Gamma; C \otimes \rho(S) \vdash \mathbf{pack}\ V : [\rho(S)] \triangleright C}
$$

3 Conclusions and Future Work

We have presented our work in progress on CGV—Capable GV. At the time of writing, we have completed syntax of types and terms as well as typing rules and we are working on finalising the operational semantics and type safety results. CGV uses capabilities in order to introduce channel sharing and provide greater expressivity. This is achieved via tracking capabilities in a type-and-effect system and making the channel endpoints unrestricted. While this approach allows sharing, it also reintroduces deadlocks. Hence, a potential area for further work will be the combination of CGV with Priority GV [6, 5] to restore deadlock-freedom and tie the system back to logic. Additionally, we aim to explore channel sharing through different means, namely via manifest sharing [1].

References

[1] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639. Springer, 2019.

[2] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[3] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019.

[4] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[5] Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. In *Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021*, pages 1–13. ACM, 2021.

[6] Wen Kokke and Ornela Dardha. Prioritise the best variation. In *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12719 of *Lecture Notes in Computer Science*, pages 100–119. Springer, 2021.

[7] Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.

[8] A. Laura Voinea, Ornela Dardha, and Simon J. Gay. Resource sharing via capability-based multiparty session types. In *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings*, volume 11918 of *Lecture Notes in Computer Science*, pages 437–455. Springer, 2019.

[9] Philip Wadler. Propositions as sessions. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 273–286. ACM, 2012.

A Context Split

Typing environments can be split according to rules in Figure 1 analogously to the work done by Vasconcelos [7].

That allows all channels to be unrestricted while preserving linearity for everything else.

$$\frac{}{\varnothing = \varnothing \circ \varnothing} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad T = tr(\rho)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : T = (\Gamma_1, x : T) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : T = \Gamma_1 \circ (\Gamma_2, x : T)}$$

Figure 1: Context split.

B Static Typing Rules

Full static typing rules are presented in Figure 3 and Figure 3.

Values typing judgements

Typing judgements for values are of the form

$$\Gamma \vdash V : T$$

which states that *Under typing environment* Γ*, term* V *is of type* T.

Computations typing judgements

Typing judgements for computations are of the form

$$\Gamma; C \vdash M : T \triangleright C'$$

which states that *Under typing environment* Γ *and with capability set* C*, term* M *is of type* T *and produces capability set* C'.

$$\textsc{T-Var}\ \frac{}{x : T \vdash x : T} \qquad \textsc{T-Unit}\ \frac{}{\varnothing \vdash () : \mathbf{1}} \qquad \textsc{T-Lam}\ \frac{\Gamma, x : T; C \vdash M : U \triangleright C'}{\Gamma \vdash \lambda x.M : T\ (C) \multimap (C')\ U}$$

$$\textsc{T-PairVal}\ \frac{\Gamma \vdash V : T \qquad \Delta \vdash W : U}{\Gamma \circ \Delta \vdash (V, W) : T \times U}$$

$$\textsc{T-Inl}\ \frac{\Gamma \vdash V : T}{\Gamma \vdash \mathbf{inl}\ V : T + U} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash V : U}{\Gamma \vdash \mathbf{inr}\ V : T + U}$$

Figure 2: Static Typing Rules for Values.

$$\textsc{T-App}\ \frac{\Gamma \vdash V : T\ (C) \multimap (C')\ U \qquad \Delta \vdash W : T}{\Gamma \circ \Delta; C \vdash V\ W : U \triangleright C'} \qquad \textsc{T-Return}\ \frac{\Gamma \vdash V : T}{\Gamma; C \vdash \mathbf{return}\ V : T \triangleright C'}$$

$$\textsc{T-LetUnit}\ \frac{\Gamma \vdash V : \mathbf{1} \qquad \Delta; C \vdash M : T \triangleright C'}{\Gamma \circ \Delta; C \vdash \mathbf{let}\ () = V\ \mathbf{in}\ M : T \triangleright C'} \qquad \textsc{T-LetPair}\ \frac{\Gamma \vdash V : T \times T' \qquad \Delta, x : T, y : T'; C \vdash M : U \triangleright C'}{\Gamma \circ \Delta; C \vdash \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M : U \triangleright C'}$$

$$\textsc{T-LetBind}\ \frac{\Gamma; C \vdash M : T \triangleright C' \qquad \Delta, x : T; C' \vdash N : U \triangleright C''}{\Gamma \circ \Delta; C \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : U \triangleright C''}$$

$$\textsc{T-CaseSum}\ \frac{\Gamma \vdash L : T + T' \qquad \Delta, x : T; C \vdash M : U \triangleright C' \qquad \Delta, y : T'; C \vdash N : U \triangleright C'}{\Gamma \circ \Delta; C \vdash \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U \triangleright C'}$$

$$\textsc{T-New}\ \frac{}{\varnothing; C \vdash \mathbf{new} : tr(\rho) \times tr(\rho') \triangleright C \otimes \rho(S) \otimes \rho'(\overline{S})}$$

$$\textsc{T-Spawn}\ \frac{\Gamma; C \vdash M : [\rho(S)] \triangleright C' \qquad \Delta; \rho(S) \vdash N : \mathbf{1} \triangleright \varnothing}{\Gamma \circ \Delta; C \vdash \mathbf{spawn}\ MN : \mathbf{1} \triangleright C'} \qquad \textsc{T-Close}\ \frac{\Gamma \vdash V : tr(\rho)}{\Gamma; C \otimes \rho(\mathbf{end}) \vdash \mathbf{close}\ V : \mathbf{1} \triangleright C}$$

$$\textsc{T-Send}\ \frac{\Gamma \vdash V : T \times tr(\rho)}{\Gamma; C \otimes \rho(!T.S) \vdash \mathbf{send}\ V : \mathbf{1} \triangleright C \otimes \rho(S)} \qquad \textsc{T-Recv}\ \frac{\Gamma \vdash V : tr(\rho)}{\Gamma; C \otimes \rho(?T.S) \vdash \mathbf{recv}\ V : T \triangleright C \otimes \rho(S)}$$

$$\textsc{T-Pack}\ \frac{\Gamma \vdash V : tr(\rho)}{\Gamma; C \otimes \rho(S) \vdash \mathbf{pack}\ V : [\rho(S)] \triangleright C} \qquad \textsc{T-Unpack}\ \frac{\Gamma \vdash V : [\rho(S)]}{\Gamma; C \vdash \mathbf{unpack}\ V : \mathbf{1} \triangleright C \otimes \rho(S)}$$

Figure 3: Static Typing Rules for Computations.

Case Study on Displayed Monoidal Categories

Benedikt Ahrens[1,2], Ralph Matthes[3], and Kobe Wullaert[1]

[1] Delft University of Technology, The Netherlands
[2] University of Birmingham, United Kingdom
[3] IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France

A new formalization of monoidal categories A monoidal structure on a category C is given by a tensor product $\otimes$, that is, a functorial binary operation on the objects and morphisms of C. Furthermore there is a unit object I, that is neutral for the tensor operation modulo (natural) isomorphism—not "on the nose", i. e., not with propositional equality. The tensor is associative up to isomorphism, and a pentagon law holds for that isomorphism, as well as a triangle law connecting all three isomorphisms.

Monoidal categories abound in mathematics (a prime example being the vector spaces over a given field) and are also well present in theoretical computer science. Functors between categories are accordingly extended to monoidal functors, so as to ensure preservation of the extra structure. The operations doing this for tensor and unit have to interact properly with the aforementioned isomorphisms. *Strong* monoidal functors need these operations to be isomorphisms, while *strict* monoidal functors satisfy the laws "on the nose". Monoidal functors abstractly capture the notion of "homomorphism" in the case of one binary operation and one constant when not only objects but also their morphisms matter (which is typical of constructive logic).

In textbooks definitions, the tensor product is seen as a bifunctor on C, i. e., a functor from $\mathsf{C} \times \mathsf{C}$ to C. Our previous attempts at defining displayed monoidal categories (see next paragraph) on that basis suffered from major difficulties with transport along components of pairs arising with the use of this product category $\mathsf{C} \times \mathsf{C}$. Instead of working with two-place functions (encoded by pairing), one can move to a curried view that first takes the left argument and then is a function that expects the right-hand side argument—which is good for the object mapping. For the two-place morphism mapping, we employ a symmetric approach, by considering the one-place mappings where the left resp. right argument is fixed to the identity, which we call the left resp. right *whiskering*, respectively. This notion is not confined to the tensor of a monoidal category but is an alternative view for any bifunctor $\mathsf{A} \times \mathsf{B} \to \mathsf{C}$. However, calling it whiskering comes from the analogous treatment of horizontal composition in bicategories in the UniMath library.

As a benefit, the formal development of monoidal categories in this format is in close correspondence with bicategories (as they are formalized in UniMath [1, Definition 2.1])—mathematically, monoidal categories are just one-object bicategories. Still, the full definition of bicategories is much heavier than the definition of monoidal categories we are obtaining, and working with one-object instances of the general bicategorical theory did not seem an option.

For lack of space, we cannot detail the other steps to getting monoidal categories and their strict or strong functors, but the readers can consult the files with the string `Whiskered` in the name in the formalization (the 1.7kloc are approximately half vernacular and half proofs, according to `coqwc`).[1]

A formalization of displayed monoidal categories A displayed category D over some base category C ([2] involving the first author) has more than just the data of a category; it reflects the construction process done on the objects and morphisms of C. However, there is a

[1] https://github.com/UniMath/UniMath/blob/1580dab0/UniMath/CategoryTheory/Monoidal

generic construction of an "ordinary" category from D, the total category $\int \mathsf{D}$, and a forgetful functor π_1 down back to C. The notion of displayed category has led to "displayed versions" of numerous categorical concepts and is seen to have potential for much more [5]. In general, "displaying" means constructing out of the ingredients of the underlying structure in a concise way avoiding copying as much as possible.

Of special interest to our application is the identification of a class of functors F from C to $\int \mathsf{D}$ that have $\pi_1 \circ F = 1_{\mathsf{C}}$. This has been formalized in the UniMath library as "sections",[2] for which such a functor F can be generically constructed. Of course, this is not formulated in terms of the total category but gives adaptations of the functor laws to hold "over" C. By working with sections, we efficiently replace the very cumbersome use of $\pi_1 \circ F = 1_{\mathsf{C}}$ as an equation between functors for rewriting (functor equality is very bad for rewrites from the point of view of intensional type theory) by definitional equality. Let us mention that displayed notions in general provide better behaviour w. r. t. equality reasoning: much less transport operations along equational hypotheses are needed than when directly working with the total categories.

Given a monoidal category C in the whiskered format and a displayed category D over (the base category of) C, we add a tensor "over" the tensor of C (as a displayed bifunctor, similarly defined in whiskered format) and add the other ingredients and laws to have a definition of displayed monoidal category D^+. A corresponding total (curried) monoidal category $\int \mathsf{D}^+$ is then obtained, with a forgetful functor π_1 back to C that we show to be strict monoidal.

Then, we identify a class of strong monoidal functors F from C to $\int \mathsf{D}^+$ s. t. $\pi_1 \circ F = 1_{\mathsf{C}}$ with a concise description in terms of D^+, which gives the notion of strong monoidal sections.

Application scenario Recent work [3, §4.3] involving the first and second authors attempted to establish a bijection between a class of parameterized distributivities (given actions in the sense of Janelidze and Kelly [4] as strong monoidal functors into some functor category) and the monoidal sections for a specially crafted displayed monoidal category (the interested reader may consult the high-level description there—the details go far beyond the capacity of the present abstract). However, in that attempt, the authors tried to work with a naive definition of monoidal sections based on the "classical" bifunctorial view of the tensor; this definition generated many problems with transport, lack of implicit argument synthesis of Coq and an unpleasant need for re-packaging tuples. Consequently, for that work, only a function in one direction could be constructed.

Using our new definition of monoidal category and the resulting workable definition of displayed monoidal category, we have been able to construct the full bijection: we have constructed the missing function, using the aforementioned monoidal sections of that displayed monoidal category as its domain, and shown that the two functions are inverse to each other (for a bicategorical generalization). We have thus solved the open question of [3, §4.3]; furthermore, the compilation time of the respective file[3] is 53% less than for the original one.[4]

Conclusion We have introduced displayed monoidal categories, with a focus on implementation and use in the UniMath library. The case study of 2.1kloc (code mostly not written from scratch but adapted from the earlier approach) validates this approach. As future work, we see basing also the notion of action-based strength on our new format for monoidal categories.

[2] https://github.com/UniMath/UniMath/blob/1580dab0/UniMath/CategoryTheory/DisplayedCats/Constructions.v#L399

[3] https://github.com/UniMath/UniMath/blob/1580dab0/UniMath/Bicategories/MonoidalCategories/ActionBasedStrongFunctorsWhiskeredMonoidal.v

[4] 43s versus 92s wall clock time measured on current Intel processor in single-thread compilation.

References

[1] Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccolò Veltri, and Niels van der Weide. Bicategories in univalent foundations. *Mathematical Structures in Computer Science*, 2022. 38 pages, online publication at https://doi.org/10.1017/S0960129522000032.

[2] Benedikt Ahrens and Peter LeFanu Lumsdaine. Displayed categories. *Log. Methods Comput. Sci.*, 15(1), 2019.

[3] Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. Implementing a category-theoretic framework for typed abstract syntax. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 307–323. ACM, 2022.

[4] George Janelidze and Gregory Maxwell Kelly. A note on actions of a monoidal category. *Theory and Applications of Categories*, 9(4):61–91, 2001. Online available at http://www.tac.mta.ca/tac/volumes/9/n4/9-04abs.html.

[5] Jonathan Sterling and Carlo Angiuli. Foundations of relative category theory. Online lecture course, available at https://www.jonmsterling.com/math/lectures/categorical-foundations.html, 2022.

Certified Abstract Machines for Skeletal Semantics

Guillaume Ambal[1], Sergueï Lenglet[2], and Alan Schmitt[3]

[1] Univ Rennes, Rennes, France
guillaume.ambal@irisa.fr
[2] Université de Lorraine, Nancy, France
serguei.lenglet@univ-lorraine.fr
[3] Inria, Rennes, France
alan.schmitt@inria.fr

Skeletal Semantics. Skeletal semantics [8] is a framework, relying on a meta-language named Skel, to formalize the operational semantics of programming languages. It is based on a general and systematic way to break down the semantics of a language. The fundamental idea is to only specify the structure of evaluation functions (e.g., sequences of operations, non-deterministic choices, recursive calls) while keeping abstract basic operations (e.g., updating an environment or comparing two values). The motivation for this semantics is that the structure can be analyzed, transformed, or certified independently from the implementation choices of the basic operations.

The OCaml [14] implementation benefits from a toolbox called Necro. It can generate an OCaml interpreter [10] or a Coq [19] mechanization [8] of a skeletal semantics given as input.

Skeletal semantics is a framework generic enough to express any semantics which can be written with inductive rules. For instance, consider a call-by-value λ-calculus with closures. The syntax $t ::= \lambda x.t \mid x \mid t\ t$ and rules of the form $s, \lambda x.t \Downarrow (x, t, s)$ can be represented as follows.

```
type ident
type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)
...
```

```
val eval (s : env) (l : lterm) : clos =
  branch
    let Lam (x, t) = l in
    Clos (x, t, s)
  or ...
  end
```

The syntax is represented by unspecified types (e.g., `ident`) and algebraic data types (e.g., `lterm`). The semantics is represented by evaluation functions (e.g., `eval`) defined using *skeletons*.

To make sense of this representation, we attribute a meaning to skeletons, called an *interpretation*. The main one, called *concrete interpretation*, is written in a non-deterministic big-step style and formalized in Coq. While useful to prove some properties of a language or of programs, the concrete interpretation cannot reason about non-terminating programs and it is quite far from an actual implementation. We thus propose alternate interpretations, in the form of non-deterministic and deterministic abstract machines, derived using functional correspondence [1]. These new interpretations are proved sound in relation to the concrete interpretation, and we use the deterministic version to generate a certified generic OCaml interpreter.

This work summarizes a paper recently accepted at CPP 2022 [4], and our results are outlined in Figure 1.

Functional Correspondence. Functional correspondence [1] is a systematic strategy for transforming functional evaluators (i.e., big-step interpreters) into equivalent abstract machines. This approach combines several known transformations. The main phases of the derivation are a CPS-transformation [17], a phase of defunctionalization [18], and then the proper creation of the abstract machine and its evaluation modes.

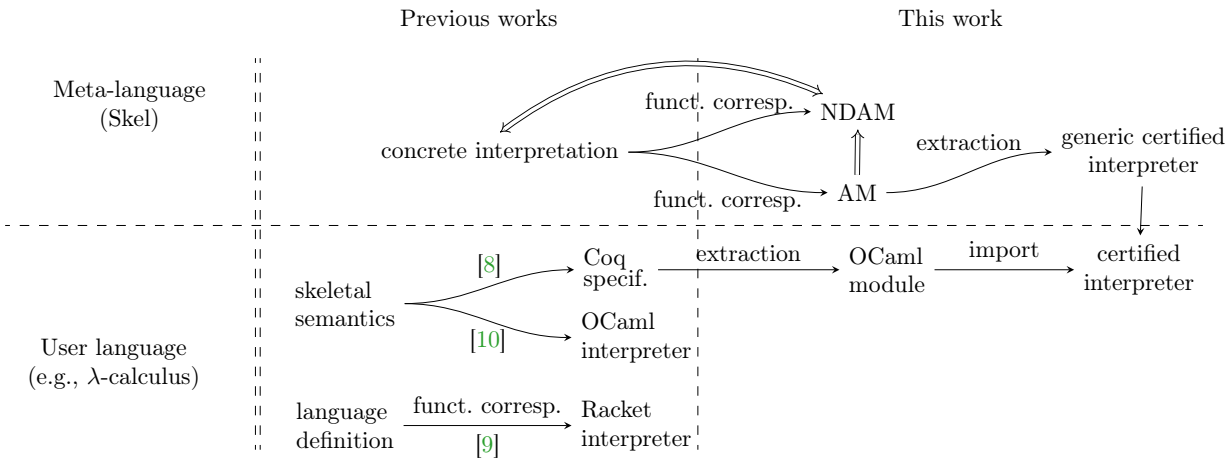


Figure 1: Summary of our work and comparison with related work

The technique of functional correspondence has been manually applied to many languages with many different features [5, 2, 16, 7, 6, 3, 12, 13], showing its robustness and usefulness. Recently, a tool was also developed for the automatic application of the technique [9].

Abstract Machines. We apply by hand the standard strategy of functional correspondence on the concrete interpretation of skeletal semantics (i.e., the big-step semantics of the meta-language Skel). Since the input semantics is non-deterministic, we obtain a non-deterministic abstract machine (NDAM) for Skel. While the transformation is classic, a novelty of our approach is to use it at the meta-level. This yields a generic abstract machine that can be proved sound once and for all, independently of the input language.

To create a deterministic executable version (AM), we proceed similarly but use a more involved CPS-transformation with two continuations [11], allowing for checkpoints and backtracking during a computation.

The concrete interpretation was already defined in the Coq proof assistant. Our two new abstract machine interpretations are formalized in Coq, and we certify them independently from the skeletal semantics (language) we are interested in. First, we prove that the NDAM is equivalent to the standard concrete interpretation. Second, the AM is proved sound with respect to the NDAM, i.e., if the AM finds a result, then the NDAM can also find the same result. The AM does not necessarily find a result, as it can get stuck in an infinite computation. By transitivity, the AM is also sound w.r.t. the concrete interpretation.

Certified Interpreter. Using the Coq extraction mechanism [15] on the deterministic abstract machine, we obtain a certified OCaml interpreter that can be instantiated with any language. From a user-defined language written as a skeletal semantics, the existing framework [8] can automatically produce the Coq deep embedding, which itself can be used to instantiate our extracted interpreter. We therefore obtain a certified interpreter for the language at no extra cost for the user.

The advantage of working at the meta-level, i.e., proving correctness once and for all languages, has a drawback: the execution happens in the meta-language, namely Skel. In contrast, the previous tool [10] produces a more efficient OCaml interpreter and allows the user to work at the level of the language, e.g., λ-terms, but without any guarantees.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19. ACM, 2003.

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.*, 90(5):223–232, 2004.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.*, 342(1):149–172, 2005.

[4] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. Certified abstract machines for skeletal semantics. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 55–67. ACM, 2022.

[5] Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. An abstract machine for strong call by value. In Bruno C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, volume 12470 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2020.

[6] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Log. Methods Comput. Sci.*, 1(2), 2005.

[7] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*, volume 3018 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2003.

[8] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.*, 3(POPL):44:1–44:31, 2019.

[9] Maciej Buszka and Dariusz Biernacki. Automating the functional correspondence between higher-order evaluators and abstract machines. In *LOPSTR 2021*, volume abs/2108.07132, 2021.

[10] Nathanaël Courant, Enzo Crance, and Alan Schmitt. Necro: Animating Skeletons. In *ML 2019*, Berlin, Germany, August 2019.

[11] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, pages 151–160. ACM, 1990.

[12] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *J. Comput. Syst. Sci.*, 76(5):302–323, 2010.

[13] Wojciech Jedynak, Malgorzata Biernacka, and Dariusz Biernacki. An operational foundation for the tactic language of coq. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 25–36. ACM, 2013.

[14] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system: Documentation and user's manual*, 2021.

[15] Pierre Letouzey. A new extraction for coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002.

[16] Maciej Piróg and Dariusz Biernacki. A systematic derivation of the STG machine verified in coq. In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 25–36. ACM, 2010.

[17] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[18] John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972.

[19] The Coq Development Team. The coq proof assistant, January 2021.

Conservativity of Two-Level Type Theory Corresponds to Staged Compilation

András Kovács

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

Two-level type theory (2LTT) [1] was originally intended as a tool to make metatheoretical reasoning about constructions in homotopy type theory more convenient, by internalizing an extensional metatheory in a syntactic layer.

2LTT as a two-stage language. First, we observe that basic variants of 2LTT formalize two-stage compilation, where the meta-level syntactic fragment contains static (compile-time) computations, and *staging* is the algorithm which performs all static computation, producing object-theoretic syntax as output. The basic rules are the following. There are two universes, U_0 and U_1, where U_0 classifies object-level (runtime) types and U_1 classifies meta-level (compile time) types. There are three *staging operations*.

Lifting: for $A : \mathsf{U}_0$, we have $\Uparrow A : \mathsf{U}_1$. From the staging point of view, $\Uparrow A$ is the type of metaprograms which compute runtime expressions of type A.

Quoting: for $A : \mathsf{U}_0$ and $t : A$, we have $\langle t \rangle : \Uparrow A$. A quoted term $\langle t \rangle$ represents the metaprogram which immediately yields t.

Splicing: for $A : \mathsf{U}_0$ and $t : \Uparrow A$, we have $\sim t : A$. During staging, the metaprogram in the splice is executed, and the resulting expression is inserted into the output.

Quoting and splicing are definitional inverses. Also, the above operations are the *only* way of crossing between stages; all type formers stay within a single stage, and in particular we cannot eliminate from one stage to a different one. The staging interpretation of 2LTT remains valid with arbitrary assumed type formers. Note that all three operations correspond to features in existing staged systems such as MetaOCaml [4] or typed Template Haskell [8], although none of the existing systems support staging with dependent types.

Conservativity as staging. By conservativity we mean the following. There is an embedding morphism $\ulcorner - \urcorner$ which maps from the object theory to the object-level syntactic fragment of 2LTT. 2LTT is conservative if $\ulcorner - \urcorner$ is bijective on types and terms, i.e. $\mathsf{Ty}_{\mathsf{Obj}}\,\Gamma \simeq \mathsf{Tm}_{\mathsf{2LTT}}\,\ulcorner\Gamma\urcorner\,\mathsf{U}_0$ and $\mathsf{Tm}_{\mathsf{Obj}}\,\Gamma\,A \simeq \mathsf{Tm}_{\mathsf{2LTT}}\,\ulcorner\Gamma\urcorner\,\ulcorner A\urcorner$.

A staging algorithm consists of functions $\mathsf{Stage} : \mathsf{Tm}_{\mathsf{2LTT}}\,\ulcorner\Gamma\urcorner\,\mathsf{U}_0 \to \mathsf{Ty}_{\mathsf{Obj}}\,\Gamma$ and $\mathsf{Stage} : \mathsf{Tm}_{\mathsf{2LTT}}\,\ulcorner\Gamma\urcorner\,\ulcorner A\urcorner \to \mathsf{Tm}_{\mathsf{Obj}}\,\Gamma\,A$. We call Stage *stable* if $\mathsf{Stage} \circ \ulcorner - \urcorner = \mathsf{id}$, and *sound* if $\ulcorner - \urcorner \circ \mathsf{Stage} = \mathsf{id}$. Hence, conservativity is the same as having a sound and stable staging algorithm. In [1], only a weak form of conservativity is shown, which corresponds to staging without soundness.

Staging by evaluation. We define Stage as the evaluation of 2LTT types and terms in the presheaf model over the syntactic category of the object theory. We call this model $\widehat{\mathsf{Obj}}$. The object-level 2LTT fragment is interpreted using sets of types and terms in the object theory. Operationally, this yields closed evaluation for the meta-level 2LTT fragment, and we get naive weakening for object-theoretic terms. Naive weakening can be inefficient, but in practice it can be optimized using De Bruijn levels and delayed variable renamings. The same efficiency issue arises in presheaf-based normalization-by-evaluation, and the same solution applies there.

However, staging can be overall more efficient, because meta-level evaluation is closed (no free variables occur in values).

Soundness. Stability of staging follows by straightforward induction on the object theory; soundness requires more effort. We define a *restriction* morphism, which maps from 2LTT to $\widehat{\mathsf{Obj}}$, restricting the meta-level syntactic fragment so that it can only depend on object-level typing contexts, that is, contexts given as $\ulcorner\Gamma\urcorner$ for some Γ. Then, we show soundness of staging by a proof-relevant logical relation between the evaluation morphism from 2LTT to $\widehat{\mathsf{Obj}}$ and the restriction morphism. We define this logical relation in the internal language of $\widehat{\mathsf{Obj}}$, to avoid the deluge of boilerplate for showing stability under object-theoretic substitution.

Alternatively, staging together with its soundness can be established in a single step, by gluing along the restriction morphism. This interpretation can be more compactly defined using synthetic Tait computability [7].

Intensional analysis. This means analyzing the internal structure of object-level terms, i.e. values with type $\Uparrow A$. While intensional analysis can be often simulated with deeply embedded inductive syntaxes at the meta level, it may be more concise and convenient to use native intensional analysis features instead. We consider the interpretation of such features in the presheaf models.

If the object theory has parallel substitutions as syntactic morphisms, then $\widehat{\mathsf{Obj}}$ does not support intensional analysis. For illustration, consider decidable equality of $\Uparrow A$ as an intensional meta-level axiom; this is essentially decidability of definitional equality of object terms. This axiom does not hold in $\widehat{\mathsf{Obj}}$, because inequality of object-level terms is not stable under substitution: inequal variables can be mapped to equal terms.

However, we may choose to only have *weakenings* as morphisms in the object syntax. In this case, decidable equaity for $\Uparrow A$ holds in $\widehat{\mathsf{Obj}}$, since term inequality is stable under weakening. As a trade-off, if there is no notion of substitution in the specification of the object theory, it is not possible to specify dependent or polymorphic types there. This is still sufficient for many practical use cases, for example when the object theory is simply-typed, in which case staging also performs *monomorphization*. In this setup, it makes sense to only have weakening in the equational theory of the object theory, but no $\beta\eta$-rules and no substitution. The reason is that we do not want to equate programs with different performance characteristics, when we do staging in order to improve runtime code performance.

Future work. A major line of future work is to connect 2LTT to existing literature on staged compilation. This would involve formalizing existing tricks and techniques, such as let-insertion techniques [5], fusion, various binding-time improvements and CPS conversions [3]. Another line is fleshing out practical details for staging and intensional analysis. In staging, a production-strength solution should include some form of caching, to reduce code duplication. In intensional analysis, some form of induction or pattern matching would be more ergonomic than plain decidability of conversion.

Also, 2LTT could be extended to more general *multimodal* [2] type theories, where modalities represent morphisms between different object-theoretic syntactic categories, in possibly different object theories. A simple example is the closed or "crisp" modality [6] which can be used to represent closed object terms.

References

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

[2] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. 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 492–506. ACM, 2020.

[3] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.

[4] Oleg Kiselyov. The design and implementation of BER metaocaml - system description. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer, 2014.

[5] Oleg Kiselyov and Jeremy Yallop. let (rec) insertion without effects, lights or magic. *CoRR*, abs/2201.00495, 2022.

[6] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal universes in models of homotopy type theory. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[7] Jonathan Sterling. *First Steps in Synthetic Tait Computability*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2021.

[8] Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. Staging with class: a specification for typed template haskell. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022.

Consistent Ultrafinitist Logic

Michał J. Gajda

Migamake Pte Ltd
mjgajda@migamake.com

Ultrafinitism(Kornai 2003; Podnieks 2005; Yessenin-Volpin 1970; Gefter 2013; Lenchner 2020) postulates that we can only reason and compute relatively short objects(Seth Lloyd 2000; Krauss and Starkman 2004; Sazonov 1995; S. Lloyd 2002; Gorelik 2010), and numbers beyond certain value are not available. Some philosophers also question the physical existence of real numbers beyond a certain level of accurracy(Gisin 2019). This approach would also forbid many forms of infinitary reasoning and allow removing many from paradoxes stemming from a countable enumeration.

However, philosophers still disagree on whether such a finitist logic could be consistent(Magidor 2007), while constructivist mathematicians claim that *"no satisfactory developments exist"*(Troelstra 1988). We present preliminary work on a proof system based on Curry-Howard isomorphism(Howard 1980) and explicit bounds for computational complexity.

This approach invalidates logical paradoxes that stem from a profligate use of transfinite reasoning(Benardete 1964; Nolan forthcoming; Schirn and Niebergall 2005), and assures that we only state problems that are decidable by the limit on input size, proof size, or the number of steps(Tarski, Mostowski, and Robinson 1955).

Consideration of complexity also solves other paradoxes, in particular the "paradox of inference" existing in classical theory of semantic information(Bar-Hillel and Carnap 1953; Duzi 2010). Using a bound on cost and depth of the term for each inference, we independently developed a very similar approach to that used for cost bounding in higher-order rewriting(Vale and Kop 2021).

By *finitism* we understand the mathematical logic that tries to absolve us from transfinite inductions(Kornai 2003). *Ultrafinitism* goes even further by postulating a definite limit for the complexity of objects that we can compute with(Seth Lloyd 2000; Krauss and Starkman 2004; Sazonov 1995; S. Lloyd 2002; Gorelik 2010). We assume these without committing to a particular limit.

In order to permit only *ultrafinitist*[1] inferences, we postulate *ultraconstructivism*: we permit proofs, or constructions that are not just strictly computable, but for which there is a bound on amount of computation that is needed to resolve them. That means that we forbid proofs that go for an arbitrarily long time and require a *deadline* for any proof or computation.

For the sake of generality, we will attach this deadline in the form of *bounding function* that takes as arguments *depths of input terms*, and outputs the upper bound on the number of steps that the proof is permitted to make. *Depths of input terms* are a convenient upper bound on the complexity of normalized proof terms (those without the cut.)

Please note that notation $\forall x_v : A \to^{\alpha(v)}_{\beta(v)} B$ has a size variable v declared as a depth of term variable x, and then bound in polynomials $\alpha(v)$ and $\beta(v)$ The notation $\alpha(1)$ is a shortcut for $\alpha[\![1/v]\!]$ in the rules *abs* and *app*.

[1] Also called *strict finitist* by (Magidor 2007).

$$
\begin{array}{lrcl}
\text{Size variables:} & v & \in & V \\
\text{Term variables:} & x & \in & X \\
\text{Positive naturals:} & i & \in & \mathbb{N} \setminus \{0\} \\
\text{Polynomials:} & \rho & ::= & v \mid i \mid \rho + \rho \mid \rho * \rho \mid \rho^{\rho} \mid iter(\rho,\rho,v) \mid \rho[\![v/\rho]\!] \\
\text{Data size bounds:} & \alpha & ::= & \rho \\
\text{Computation bounds:} & \beta & ::= & \rho \\
\text{Types:} & \tau & ::= & v \mid \tau \wedge \tau \mid \tau \vee \tau \mid \forall x_v : \tau \to^{\alpha}_{\beta} \tau \mid \bot \mid \circ \\
\text{Terms:} & E & ::= & v \mid \lambda v.E \mid in_r(E) \mid in_l(E) \mid (E,E) \mid () \\
 & & \mid & case\, E\, of \begin{array}{lcl} in_l(v) & \to & E; \\ in_r(v) & \to & E; \end{array} \\
\text{Environments:} & \Gamma & ::= & v_1 : {\tau^1}_{\beta_1}, ..., {\tau^n}_{\beta_n} \\
\text{Judgements:} & J & ::= & \Gamma \vdash^{\alpha}_{\beta} E : \tau
\end{array}
$$

$$\frac{\Gamma \vdash^{\alpha}_{\beta} y_{\beta} : A \quad v \in V}{\Gamma, x_{\beta} : A \vdash^{1}_{\beta} x : A}\ var \qquad \frac{}{\Gamma \vdash^{1}_{\beta} () : \circ}\ unit$$

$$\frac{\Gamma \vdash^{\alpha_1}_{\beta_1} a^1 : A^1 \quad \Gamma \vdash^{\alpha_2}_{\beta_2} a^2 : A^2}{\Gamma \vdash^{\alpha_1+\alpha_2}_{\max(\beta_1,\beta_2)+1} (a^1, a^2) : A^1 \wedge A^2}\ pair \qquad \frac{\Gamma \vdash^{\alpha}_{\max(\beta_1,\beta_2)} e : A^1 \wedge A^2 \quad i \in \{1,2\}}{\Gamma \vdash^{\alpha+1}_{\beta-1} prj_i\ e : A^i}\ prj_i$$

$$\frac{\Gamma \vdash^{\alpha}_{\beta} e : A^i \quad i \in \{l,r\}}{\Gamma \vdash^{\alpha+1}_{\beta+1} in_i(e) : A^1 \vee A^2}\ inj \qquad \frac{\Gamma \vdash^{\alpha_1}_{\beta_1} e : A \qquad \alpha_1 \le \alpha_2 \qquad \beta_1 \le \beta_2}{\Gamma \vdash^{\alpha_2}_{\beta_2} e : A}\ subsume$$

$$\frac{\Gamma \vdash^{\alpha_\vee}_{\beta_\vee} a : A^1 \vee A^2 \qquad \Gamma, x : {A^1}_{\beta_\vee - 1} \vdash^{\alpha_1}_{\beta_1} b : B \qquad \Gamma, y : {A^2}_{\beta_\vee - 1} \vdash^{\alpha_2}_{\beta_2} c : B}{\Gamma \vdash^{\alpha_\vee + max(\alpha_1,\alpha_2)+1}_{\max(\beta_1,\beta_2)} case\, a\, of \begin{array}{lcl} in_l(x) & \to & b; \\ in_r(y) & \to & c; \end{array} : B}\ case$$

$$\frac{\Gamma, x_v : A \vdash^{\alpha(v)}_{\beta(v)} e : B}{\Gamma \vdash^{\alpha(1)+1}_{\beta(1)+1} \lambda x.e : \forall a_v : A \to^{\alpha(v)}_{\beta(v)} B}\ abs \qquad \frac{\Gamma \vdash^{\alpha_1}_{\beta_1} e : \forall a : A_v \to^{\alpha_2(v)}_{\beta_2(v)} B \quad \Gamma \vdash^{\alpha_3}_{\beta_3} a : A}{\Gamma \vdash^{\alpha_1+\alpha_2(\beta_3)+\alpha_3}_{\beta_2(\beta_3)} e\ a : B}\ app$$

$$\frac{\Gamma \vdash^{\alpha_1}_{\beta_1} f : A_v \to^{\alpha_2(v_1)}_{\beta_2(v_2)} A \quad \Gamma \vdash^{\alpha_3}_{\beta_3} k : B \quad \Gamma \vdash^{\alpha_4}_{\beta_4} a : A}{\Gamma \vdash^{\alpha_1+\alpha_3+iter(\alpha_2,\beta_3,v_1)[\![\beta_4/v_1]\!]+\alpha_4}_{\beta_1[\![iter(\beta_2,\beta_3,v_2)[\![\beta_4/v_2]\!]/v]\!]} rec(f,k,a) : B}\ rec$$

After elision of bounds and rule *subsume* we see the rules for intuitionistic logic. Thus consistency can be proved by the consistency of intuitionistic logic(Brouwer 1981; Van Dalen 1986; Sørensen and Urzyczyn 1998). Every valid proposition with a *fixed bound on input n* can be checked by enumerating inputs, and is thus decidable. It is easy to show that our logic can emulate bounded loop programs(Meyer and Ritchie 1967) which have power equivalent to primitive recursive functions(Robinson 1947). Expressing statements about undecidability implicitly requires unbounded computational effort. Since all our proofs and arguments are explicitly bounded, there is no room to state undecidability. We thus define statements that are both true, and computable a given limit(Gorelik 2010).

References

Bar-Hillel, Yehoshua, and Rudolf Carnap. 1953. "Semantic Information." *British Journal for the Philosophy of Science* 4 (14): 147–57. https://doi.org/10.1093/bjps/IV.14.147.

Benardete, Jose. 1964. *Infinity: An Essay in Metaphysics*. Clarendon Press.

Brouwer, L. E. J. 1981. *Over de Grondslagen Der Wiskunde*. Vol. 1. MC Varia. Amsterdam: Mathematisch Centrum.

Duzi, Marie. 2010. "The Paradox of Inference and the Non-Triviality of Analytic Information." *Journal of Philosophical Logic* 39 (October): 473–510. https://doi.org/10.1007/s10992-010-9127-5.

Gefter, Amanda. 2013. "Mind-Bending Mathematics: Why Infinity Has to Go." *New Scientist* 219 (2930): 32–35. https://doi.org/https://doi.org/10.1016/S0262-4079(13)62043-6.

Gisin, Nicolas. 2019. "Indeterminism in Physics, Classical Chaos and Bohmian Mechanics. Are Real Numbers Really Real?" https://arxiv.org/abs/1803.06824.

Gorelik, Gennady. 2010. "Bremermann's Limit and cGh-Physics." https://arxiv.org/abs/0910.3424.

Howard, William A. 1980. "The Formulae-as-Types Notion of Construction." In *To h.b. Curry: Essays on Combinatory Logic, λ-Calculus and Formalism*, edited by J. Hindley and J. Seldin, 479–90. Academic Press.

Kornai, Andras. 2003. "Explicit Finitism." *International Journal of Theoretical Physics* 42 (February): 301–7. https://doi.org/10.1023/A:1024451401255.

Krauss, Lawrence, and Glenn Starkman. 2004. "Universal Limits on Computation," May.

Lenchner, Jonathan. 2020. "A Finitist's Manifesto: Do We Need to Reformulate the Foundations of Mathematics?" https://arxiv.org/abs/2009.06485.

Lloyd, S. 2002. "Computational Capacity of the Universe." *Physical Review Letters* 88 23: 237901.

Lloyd, Seth. 2000. "Ultimate Physical Limits to Computation." *Nature* 406 (6799): 1047–54. https://doi.org/10.1038/35023282.

Magidor, Ofra. 2007. "Strict Finitism Refuted?" *Proceedings of the Aristotelian Society* 107 (1pt3): 403–11. https://doi.org/10.1111/j.1467-9264.2007.00230.x.

Meyer, Albert R., and Dennis M. Ritchie. 1967. "The Complexity of Loop Programs." In *Proceedings of the 1967 22nd National Conference*, 465–69. ACM '67. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/800196.806014.

Nolan, Daniel. forthcoming. "Send in the Clowns." In *Oxford Studies in Metaphysics*, edited by Karen Bennett and Dean Zimmerman. Oxford: Oxford University Press.

Podnieks, Karlis. 2005. "Towards a Real Finitism?" 2005. http://www.ltn.lv/~podnieks/finitism.htm.

Robinson, Raphael M. 1947. "Primitive recursive functions." *Bulletin of the American Mathematical Society* 53 (10): p. 925–942. https://doi.org/bams/1183511140.

Sazonov, Vladimir Yu. 1995. "On Feasible Numbers." In *Logic and Computational Complexity*, edited by Daniel Leivant, 30–51. Berlin, Heidelberg: Springer Berlin Heidelberg.

Schirn, Matthias, and Karl-Georg Niebergall. 2005. "Finitism = PRA? On a Thesis of w. W. Tait." *Reports on Mathematical Logic*, January.

Sørensen, Morten Heine B., and Pawel Urzyczyn. 1998. "Lectures on the Curry-Howard Isomorphism." http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7385.

Tarski, Alfred, Andrzej Mostowski, and Raphael M. Robinson. 1955. "Undecidable Theories." *Philosophy* 30 (114): 278–79.

Troelstra, A. S. 1988. *Constructivism in Mathematics: An Introduction*. Elsevier. https://www.sciencedirect.com/bookseries/studies-in-logic-and-the-foundations-of-mathematics/vol/121/suppl/C.

Vale, Deivid, and Cynthia Kop. 2021. "Tuple Interpretations for Higher-Order Rewriting." *CoRR* abs/2105.01112. https://arxiv.org/abs/2105.01112.

Van Dalen, Dirk. 1986. "Intuitionistic Logic." In *Handbook of Philosophical Logic: Volume III: Alternatives in Classical Logic*, edited by D. Gabbay and F. Guenthner, 225–339. Dordrecht: Springer Netherlands. https://doi.org/10.1007/978-94-009-5203-4_4.

Yessenin-Volpin, Aleksandr S. 1970. "The Ultra-Intuitionistic Criticism and the Antitraditional Program for Foundations of Mathematics." In *Studies in Logic and the Foundations of Mathematics*, 60:3–45. Elsevier.

Cubical Models are Cofreely Parametric

Hugo Moeneclaey

Université de Paris,
Inria Paris, CNRS, IRIF, France
moeneclaey@irif.fr

Parametricity was originally introduced as a syntactic property [5], showing that terms of system F treat type input uniformly. It can be extended to the syntax of type theory [2]. We adopt a semantical point of view, so we define a parametric model of type theory as a model where:

- Any type comes with a chosen relation.
- Any term preserves these relations.
- This structure obeys equations, defining inductively parametricty on type and term constructors.

So being parametric is an additional structure on a model, witnessing some kind of uniformity for its terms. The usual syntactic parametricity can be recast by saying that the initial model is parametric. Many variants of parametricity have been studied, for example:

- Realizability, where every type comes with a predicate rather than a relation.
- Internal parametricity, where every type comes with a *reflexive* relation.

In this work we use clans as models of type theory (although everything can be adapted to lex categories or even plain categories). We prove that the category of clans is symmetric monoidal closed. Then we define notions of parametricity as monoids in this category, i.e. as monoidal models. We unfold this definition:

Lemma 1. *A notion of parametricity consists of a clan with a monoidal product such that:*

- *The monoidal product commutes with finite limits in both variables.*
- *Given fibrations $i \twoheadrightarrow i'$ and $j \twoheadrightarrow j'$ we have an induced fibration:*

$$i \otimes j \twoheadrightarrow (i' \otimes j) \underset{i' \otimes j'}{\times} (i \otimes j') \tag{1}$$

Then we define a parametric model for a notion of parametricity $\mathcal{M}$ simply as an $\mathcal{M}$-module. We give the following result:

Proposition 2. *Assume $\mathcal{U}$ a symmetric monoidal closed category with $\mathcal{M}$ a monoid in $\mathcal{U}$. Then the forgetful functor from $\mathcal{M}$-modules to $\mathcal{U}$ has:*

- *A left adjoint sending $\mathcal{C}$ to:*

$$\mathcal{M} \otimes \mathcal{C} \tag{2}$$

with $\mathcal{M}$ acting through the canonical left action of $\mathcal{M}$ on itself.

- *A right adjoint sending $\mathcal{C}$ to:*

$$\mathcal{M} \multimap \mathcal{C} \tag{3}$$

with $\mathcal{M}$ acting through the canonical right action of $\mathcal{M}$ on itself.

As a corollary, we have the following for any notion of parametricity:

Theorem 1. *The forgetful functor from parametric models to arbitrary ones has left and right adjoints, and we have compact descriptions for them.*

This extends the situation from [4] to any variant of parametricity, at the cost of forgetting arrow types and universes.

Then we give many examples of notions of parametricity. We build the following using the aforementioned right adjoints, for several variants of cubical objects:

- Categories of cubical objects.
- Lex categories of truncated cubical objects.
- Clans of Reedy fibrant cubical objects.

As an example, consider $\square$ the Reedy category of cubes with faces and reflexivities only. We have the following:

Proposition 3. *There exists a monoidal clan $\widehat{\square}$ such that for any clan $\mathcal{C}$, the clan:*

$$\widehat{\square} \multimap \mathcal{C} \tag{4}$$

is equivalent to the clan of Reedy fibrant functors from $\square$ to $\mathcal{C}$.

These results mean that many cubical models are cofreely parametric, giving a solid theoretical grounding to the observation that (variant of) cubical structures arise naturally when working with (variant of) parametricity [1, 3].

References

[1] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electronic Notes in Theoretical Computer Science*, 2015.

[2] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, 2010.

[3] Evan Cavallo and Robert Harper. Internal parametricity for cubical type theory. In *28th EACSL Annual Conference on Computer Science Logic*, 2020.

[4] Hugo Moeneclaey. Parametricity and semi-cubical types. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science*, 2021.

[5] John C Reynolds. Types, abstraction and parametric polymorphism. In *Proceedings of the IFIP 9th World Computer Congres*, 1983.

Data Types with Negation (Talk Abstract)

Robert Atkey `robert.atkey@strath.ac.uk`

University of Strathclyde*, Glasgow, UK

1 Introduction

Inductive data types are a foundational tool for representing data and knowledge in dependently typed programming languages. The user provides a collection of rules that determine positive evidence for membership in the type. Elimination of an inductive type corresponds to structural induction on its members. For example, in a language like Agda we can define inductive data types for representing evidence that natural numbers are odd or even:

$$\begin{array}{l}\textbf{data}\,\mathrm{Even} : \mathrm{Nat} \to \mathrm{Set}\,\textbf{where}\\ \quad \mathsf{zero\text{-}even} : \mathrm{Even}\,\mathsf{zero}\\ \quad \mathsf{succ\text{-}odd} : \forall x \to \mathrm{Odd}\,x \to \mathrm{Even}\,(\mathsf{succ}\,x)\end{array} \qquad \begin{array}{l}\textbf{data}\,\mathrm{Odd} : \mathrm{Nat} \to \mathrm{Set}\,\textbf{where}\\ \quad \mathsf{succ\text{-}even} : \forall x \to \mathrm{Even}\,x \to \mathrm{Odd}\,(\mathsf{succ}\,x)\end{array}$$

But what if our data modelling requires negative information as well as positive? One example is this alternative encoding of evenness, which marks a number as even if its predecessor is not:

$$\begin{array}{l}\textbf{data}\,\mathrm{Even} : \mathrm{Nat} \to \mathrm{Set}\,\textbf{where}\\ \quad \mathsf{zero\text{-}even} : \mathrm{Even}\,\mathsf{zero}\\ \quad \mathsf{succ\text{-}not} : \forall x \to \neg(\mathrm{Even}\,x) \to \mathrm{Even}\,(\mathsf{succ}\,x)\end{array}$$

We have used a negation operator $\neg(-)$ to represent the negation of $\mathrm{Even}\,x$ instead of an explicit definition of oddness. A example that makes more essential use of negation is a data type representing successful parses from a backtracking Parsing Expression Grammar (PEG) parser. PEGs use an ordered choice operator `A / B` which means "try to parse as `A`, and if that fails then try to parse a `B`". If we represent evidence of successful parses as elements of a data type indexed by the string to be parsed and the suffix remaining to be parsed, then we could use a data type with negation to represent a PEG production of the form `P ::= A / B`:

$$\begin{array}{l}\textbf{data}\,\mathrm{ParseP} : \mathrm{String} \to \mathrm{String} \to \mathrm{Set}\,\textbf{where}\\ \quad \mathsf{choice1} : \forall i\,o.\,\mathrm{ParseA}\,i\,o \to \mathrm{ParseP}\,i\,o\\ \quad \mathsf{choice2} : \forall i\,o.\,\neg(\exists o'.\,\mathrm{ParseA}\,i\,o') \to \mathrm{ParseB}\,i\,o \to \mathrm{ParseP}\,i\,o\end{array}$$

So a $\mathrm{ParseP}\,i\,o$ is constructible if a $\mathrm{ParseA}\,i\,o$ is possible or if that is *not* possible for any left over o' and a $\mathrm{ParseB}\,i\,o$ is possible.

If we interpret negation as $\neg A = A \to \bot$ then a proof assistant like Agda will reject these definitions. And for good reason: a data type using this kind of negation does not define a monotone operator on sets of evidence, and so does not have a well defined least fixpoint. However, it is possible to give these types a reasonable semantics. We will take two steps. First, we need a constructive notion of negation, so that we have explicit evidence of refutations to work with. Second, we need to determine the semantics of recursive negation, which we will do using the 3-valued stable model semantics of logic programming.

*Research funded by EPSRC Grant EP/T026960/1 *AISEC: AI Secure and Explainable by Construction.*

2 Constructive Negation

To give a constructive semantics of negation, we will construct evidence for refutations of propositions simultaneously with evidence for their proofs. Negation of a proposition is then nothing more than the swapping of the status of proofs and refutations. Concretely, we define $\mathrm{Set}^{\pm} = \mathrm{Set} \times \mathrm{Set}$ with the following constructions of propositions, directly witnessing the de Morgan dualities from classical logic:

$$(A^+, A^-) \times (B^+, B^-) = (A^+ \times B^+, A^- + B^-) \qquad \Sigma x : X.\, (A^+[x], A^-[x]) = (\Sigma x : X\, A^+[x], \Pi x : X.\, A^-[x])$$
$$(A^+, A^-) + (B^+, B^-) = (A^+ + B^+, A^- \times B^-) \qquad \neg(A^+, A^-) = (A^-, A^+)$$

Entailment in $\mathrm{Set}^{\pm}$ goes forwards in the first component and backwards in the second (so we are working in the category $\mathrm{Set} \times \mathrm{Set}^{op}$, a simple form of Chu Spaces [1], a model of Linear Logic).

We can construct models of inductive data types in $\mathrm{Set}^{\pm}$ for positive functors F, where the proof and refutation parts are independent, by taking the initial algebra in the first component and the final coalgebra in the second. However, this does not immediately help us with data types that involve negation.

3 Stable Semantics of Negation

To model data types with negation, we turn to ideas from Logic Programming, where the semantics of logic programs that involve negation has been the subject of intense study. One popular semantics is well-founded or 3-valued stable semantics. Intuitively, this semantics interprets negation in terms of lack of support for a proposition given the rules defining the program. This semantics has a fixed point characterisation [2] that we can replicate in terms of data types as follows.

If we have a data type with negation (assuming no indexing for now), we can derive an operator $F(A^+, A^-) = (F^+(A^+, A^-), F^-(A^+, A^-))$ using the interpretations of the connectives given above. This does not necessarily have a fixed point in $\mathrm{Set}^{\pm}$. However, if we assume that we have some fixed (X^+, X^-) we can relatise F to X to give a functor $(F/X)(A^+, A^-) = (F^+(A^+, X^-), F^-(X^+, A^-))$, which makes the proofs and refutations independent, and so does have a least fixpoint $\mu F/X$ in $\mathrm{Set}^{\pm}$. This construction is functorial in X *covariantly* in both components, so there is a least fixpoint in $\mathrm{Set} \times \mathrm{Set}$, yielding the overall semantics:

$$\mu(X^+, X^-).(\mu A^+.\, F^+(A^+, X^-), \nu A^-.\, F^-(X^+, A^-))$$

Intuitively, we build up the semantics of a data type in stages. We start with no negative information about proofs or refutations, and derive all the positive proofs and refutations we can from this. Swapping these proofs and refutations, we use them as the negative information in the next round, and so on.

It is now possible to compute models for all data types with negation. One example is the "liar" type:

$$\textbf{data}\ \mathrm{Liar} : \mathrm{Set}\ \textbf{where}\ \mathsf{liar} : \neg\mathrm{Liar} \to \mathrm{Liar}$$

This type is assigned the model $(\bot, \bot)$ – there are no proofs and no refutations of this type.

We have now built a plausible semantics for data types with negation. It remains to be seen how best to accomplish reasoning over inhabitants of data types with negative information.

References

[1] Po-Hsiang Chu (1978): *Constructing $*$-autonomous categories.* Master's thesis, McGill University.

[2] Teodor C. Przymusinski (1990): *The Well-Founded Semantics Coincides with the Three-Valued Stable Semantics.* Fundam. Inform. 13(4), pp. 445–463.

Decidability and Semidecidability via Ordinals*

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

Constructive notions of ordinals. Ordinal numbers, or simply *ordinals*, can be seen as a generalisation of the natural numbers. They allow the characterisation of possibly infinite orders, and termination proofs are one of their main applications. There are various formulations of (initial segments of) ordinals that coincide in a classical setting, but have very different behaviour constructively. One such formulation is the ordinal notation system of Cantor normal forms (binary trees with a certain property; different presentations are presented in [FXG20]). Another formulation are set-theoretic ordinals (extensional, transitive, wellfounded orders; developed in [Uni13, Chp 10.3] and [Esc21]). Yet another formulation are Brouwer ordinal trees (constructed with zero, successors, and limits; cf. Kleene's $\mathcal{O}$ [Chu38, Kle38]). We have compared these three formulations at TYPES'21 [KNFX21b], and a full publication is now available [KNFX21a].

Decidability and partial decidability of properties of ordinals. As a rule of thumb, Cantor normal forms have decidable properties as long as we only consider finitely many of them: We can always check whether two given trees are equal, or which of the two represents the larger ordinal, and so on. It is similarly clear that properties of set-theoretic ordinals are generally undecidable: If P is proposition (or *subsingleton*, i.e. a type with at most one element), then the empty order on P is a set-theoretic ordinal. Being able to decide whether this ordinal P is equal to 1 directly corresponds to the law of excluded middle. Brouwer ordinal trees sit in the "sweet spot" in the middle: some of their properties are fully decidable, but many are "partially" decidable. In this work, we explore how this can be expressed and which statements can be proved internally in homotopy type theory.

Brouwer ordinal trees. Of course, how much is decidable about Brouwer ordinal trees depends on how exactly we define them. A very basic version is the inductive type $\mathcal{O}$ with constructors $\mathsf{zero} : \mathcal{O}$, $\mathsf{suc} : \mathcal{O} \to \mathcal{O}$, and $\mathsf{sup} : (\mathbb{N} \to \mathcal{O}) \to \mathcal{O}$, as sometimes considered in the functional programming community. This type is however too simplistic for our purposes and, unlike the other discussed formulations of ordinals, does not even come with the "correct" notion of equality; if $s : \mathbb{N} \to \mathcal{O}$ is a sequence, then $\mathsf{sup}(s_0, s_1, s_2, \ldots)$ and $\mathsf{sup}(s_1, s_0, s_2, \ldots)$ are different trees. In [KNFX21a], we construct the type $\mathsf{Brw} : \mathsf{hSet}$ mutually with a relation $\leq : \mathsf{Brw} \to \mathsf{Brw} \to \mathsf{hProp}$ *quotient inductive-inductively* [ACD+18]. The constructors for Brw are $\mathsf{zero} : \mathsf{Brw}$, $\mathsf{succ} : \mathsf{Brw} \to \mathsf{Brw}$, and $\mathsf{limit} : (\mathbb{N} \xrightarrow{<} \mathsf{Brw}) \to \mathsf{Brw}$, where $\mathbb{N} \xrightarrow{<} \mathsf{Brw}$ are *strictly* increasing sequences. In addition, we have a path constructor $\mathsf{bisim} : f \approx^{\leq} g \to \mathsf{limit}\, f = \mathsf{limit}\, g$, where $f \approx^{\leq} g$ means that f and g are bisimilar sequences.

Decidable properties of Brouwer trees. The notion of *decidability* is standard: We say that a proposition P is decidable if we have $P + \neg P$, and a predicate $R : \mathsf{Brw} \to \mathsf{hProp}$ is decidable if every $R(x)$ is decidable.

*Supported by the Royal Society, grant reference URF\R1\191055, and the UK National Physical Laboratory Measurement Fellowship project *Dependent types for trustworthy tools*.

Lemma 1. *It is decidable whether $x : \mathsf{Brw}$ is finite, i.e. whether it is in the image of the obvious map $\mathbb{N} \to \mathsf{Brw}$. If n is a finite ordinal, then equality with n and comparisons with n are decidable; i.e. the predicates $(\_ = n)$, $(\_ > n)$, and $(\_ < n)$ are decidable. Similarly, the predicates $(\_ < \omega)$ and $(\_ \geq \omega)$ are decidable.*

Proof. This is a consequence of the fact that Brw satisfies the classification property of [KNFX21a]: For a given ordinal, we can check whether it is zero, a successor, or a limit, and a limit cannot be finite. Note that this argument crucially relies on the fact that our limit constructor takes *strictly* increasing sequences as argument. Without this requirement, all of the properties mentioned in the lemma would be incorrect. □

Note that, while $(\_ \geq \omega)$ is decidable, being able to decide $(\_ > \omega)$ is equivalent to LPO, which states: $\forall f : \mathbb{N} \to \mathbf{2}.(\exists n.f_n = 1) \vee (\forall n.f_n = 0)$.

Partial decidability. While internal decidability is standard, other notions of synthetic computability theory (cf. [BR87, Bau06, FKS19]) are not used as much. In our work, we consider three formulations of partial decidability. Let P be a proposition:

1. We say that P is *semidecidable* if $\exists \alpha : \mathbb{N} \to \mathsf{Bool}.(\exists i.\alpha_i = 1) \leftrightarrow P$. This definition is due to Bauer [Bau06] (see also [EK17, FLWD+18, dJ22]).
2. Using the free ω-complete partial order on the unit type [ADK17], equivalent to the *Sierpinski type* $\mathcal{S}$ [Vel17, CUV19], one can define P to be *Sierpinski-semidecidable* if $\exists s : \mathcal{S}.(s = \top) \leftrightarrow P$. This notion of semidecidability has been used by Gilbert [Gil17].
3. If $x : \mathsf{Brw}$ is an ordinal, we can say that P *can be decided in x steps* if $\exists y : \mathsf{Brw}.(y > x) \leftrightarrow P$.

Lemma 2. *P is semidecidable if and only if it can be decided in ω steps. If P is semidecidable, then it is also Sierpinski-semidecidable.*

Proof. It is easy to translate between *semidecidable* and *decidable in ω steps*. Given a binary sequence α, we define an increasing sequence $f : \mathbb{N} \to \mathsf{Brw}$ by $f_0 \equiv \mathsf{zero}$, $f_{n+1} \equiv f_n + \omega^{\alpha_n}$. Similarly, given $x : \mathsf{Brw}$, we define a binary sequence α as follows: if x is finite, then α is constantly 0. Otherwise, if x is a successor (and infinite), α is constantly 1. The remaining case is that x is $\mathsf{limit}(f)$. If f_n is finite, we define α_n to be 0, else 1. In both cases, we have $(\exists i.\alpha_i = 1) \leftrightarrow (x > \omega)$.

In the same way, given a binary sequence α, we immediately get $s : \mathcal{S}$ by taking the least upper bound of α. □

A problem that may separate semidecidability and Sierpinski-semidecidability is the following:

Proposition 3. *Given $x : \mathsf{Brw}$ and $n, k : \mathbb{N}$, it is Sierpinski-semidecidable whether $x > \omega \cdot n + k$.*

Proof. We define $s_{\omega \cdot n + k}(x) : \mathcal{S}$ by induction on x and transfinite induction on $\omega \cdot n + k$. The most interesting case is $s_{\omega \cdot (n+1)}(\mathsf{limit}(f))$, which we define as the least upper bound of the sequence $i \mapsto s_{\omega \cdot n}(f_i)$. This uses that the limit of a sequence f is, for any i, at least $f_i + \omega$. □

We expect, but have not proved, that $x > \omega \cdot n + k$ cannot be decided in ω steps. Finally, we give examples for properties that are decidable in a number of steps different from ω:

Proposition 4. *The following three statements are equivalent: (1) P is decidable; (2) P is decidable in 0 steps; (3) P is decidable in a finite number of steps.* □

Proposition 5. *The twin prime conjecture is decidable in ω^3 steps. More generally, if $P : \mathbb{N} \to \mathsf{hProp}$ is semidecidable and $P(n+1)$ implies $P(n)$, then $\forall n.P(n)$ is decidable in ω^3 steps.* □

In the future, we hope to explore internal semidecidability and its applications further.

References

[ACD+18] 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 (FoSSaCS 2018)*, pages 293–310. Springer, Cham, 2018.

[ADK17] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2017)*, pages 534–549. Springer, 2017.

[Bau06] Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006.

[BR87] Douglas Bridges and Fred Richman. *Varieties of constructive mathematics*, volume 97. Cambridge University Press, 1987.

[Chu38] Alonzo Church. The constructive second number class. *Bulletin of the American Mathematical Society*, 44:224–232, 1938.

[CUV19] James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the delay monad by weak bisimilarity. *Mathematical Structures in Computer Science*, 29(1):67–92, 2019.

[dJ22] Tom de Jong. Agda development on constructive taboos surrounding semidecidability, 2022. Available at `cs.bham.ac.uk/~mhe/TypeTopology/SemiDecidable.html`.

[EK17] Martín H Escardó and Cory M Knapp. Partial elements and recursion via dominances in univalent type theory. In *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017.

[Esc21] Martín Escardó. Agda implementation: Ordinals, Since 2010–2021. `https://www.cs.bham.ac.uk/~mhe/TypeTopology/Ordinals.html`.

[FKS19] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*, pages 38–51, 2019.

[FLWD+18] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Marc Hermes, Dominik Kirst, Mark Koch, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. Coq library of undecidability proofs, since 2018. Available online at github.com/uds-psl/coq-library-undecidability, see especially the file theories/Synthetic/Definitions.v.

[FXG20] Fredrik Nordvall Forsberg, Chuangjie Xu, and Neil Ghani. Three equivalent ordinal notation systems in cubical Agda. In *9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pages 172–185, 2020.

[Gil17] Gaëtan Gilbert. Formalising real numbers in homotopy type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 112–124, 2017.

[Kle38] S. C. Kleene. On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(4):150–155, 1938.

[KNFX21a] Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. Connecting constructive notions of ordinals in homotopy type theory. In *46th International Symposium on Mathematical Foundations of Computer Science (MFCS 2021)*, volume 202 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 70:1–70:16, 2021.

[KNFX21b] Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. Constructive notions of ordinals in homotopy type theory, 2021. Abstract, presented at TYPES'21.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book/`, Institute for Advanced Study, 2013.

[Vel17] Niccolò Veltri. *A type-theoretical study of nontermination*. PhD thesis, Tallinn University of Technology, 2017.

Deciding the cofibration logic of cartesian cubical type theories

Robert Rose[2], Matthew Z. Weaver[3], and Daniel R. Licata[1]

[1] Wesleyan University, Middletown, Connecticut, USA
dlicata@wesleyan.edu
[2] Princeton University, Princeton, New Jersey, USA
mzw@cs.princeton.edu
[3] Wesleyan University, Middletown, Connecticut, USA
rrose@wesleyan.edu

Abstract

The defining feature of Homotopy Type Theory is the univalence principle, which identifies witnesses of propositional type equality with type equivalences. But because the standard version of HoTT [6] lacks a computational interpretation of this principle, cubical type theories were developed to remedy this situation. In HoTT, witnesses of propositional type equality are viewed as paths in a topological space. In turn, cartesian cubical type theories realize these paths as functions out of a special interval type $\mathbb{I}$. It may be understood as a synthetic analogue of the real interval $[0, 1]$.

At minimum, $\mathbb{I}$ supports reasoning about endpoints 0 and 1 and arbitrary elements x_i. It may additionally support reasoning about the max (i.e., join), min (i.e., meet), and complements of such elements. With all of these operations, $\mathbb{I}$ may be characterized as a free DeMorgan algebra on the x_i. With just max and min, $\mathbb{I}$ may be characterized as a free distributive lattice on the x_i. (For more on the design space of $\mathbb{I}$, see [2])

In cubical type theories, the structure of $\mathbb{I}$ is used to specify and reason about select portions of paths. This facility is essential to the definition of a cartesian cubical type theory. Portions of paths are described using a fragment of intuitionistic first-order equational logic over the terms of $\mathbb{I}$. Formulas of this language are called cofibrations. For example, in the context of a single variable x, $(x = 0) \vee (x = 1)$ is a cofibration which would select just the endpoints of a one-dimensional path. In the context of two variables x and y, $(x = y)$ is a cofibration which would select the diagonal of a two-dimensional cubical path (i.e., a solid square). Constraints on which equations are allowed (typically described as a choice in *generating* cofibrations) also contribute to variety in the design space of cubical type theories.

Given the goal of providing a computational interpretation of HoTT, it is of course crucial that the cofibration logic be decidable. Moreover, in the pursuit of implementation of cubical type theories, it is no less important that the decision procedure be practical. It is known that deciding cofibration entailment is coNP-hard [4]. But if this problem is coNP-complete for a particular cubical type theory, we consider this to be very good news. The size of the problems typically handled by proof assistants would be handled quickly by modern SAT solvers. Our focus therefore is on the availability of efficient reductions to the tautology problem for Boolean DNF formulas.

The current project emerged from the search for a practical decision procedure for a cubical type theory with a rich structure on $\mathbb{I}$—that of a free distributive lattice, coupled with an unconstrained notion of equational cofibration. This cubical structure is used in a cubical variant [7] of a synthetic theory of $(\infty, 1)$-categories [5], and the work in this talk was motivated by the problem of deciding judgmental equality in an implementation of this directed type theory. Entailment in this logic represents one of the hardest problems which arise from the cofibration logics of cartesian cubical type theories.

We compare this situation with that of two other cubical type theories whose implementation in proof assistants is active and ongoing. For [3], the interval structure is rich—i.e.,

that of a free DeMorgan algebra—but the equations are restricted to those which relate an arbitrary term in $\mathbb{I}$ to an endpoint, 0 or 1. This allows for an efficient factoring of an equation in a cofibration formula into a term over cofibrations of the form $(x = 0)$ or $(x = 1)$, which are join-prime. From here, the naive algorithm is to convert the cofibrations being compared into join-normal form, which has exponential run-time, and to factor the problem into ones whose antecedents are join-prime and to decide each individually, which can be done efficiently. For [1], the equations are unconstrained but the interval structure is minimal, with the result that the equations are always of the form $(x = 0)$, $(x = 1)$, or $(x = y)$, which are already join-prime. Hence the naive algorithm just described applies here as well.

Returning to the cubical type theory whose interval $\mathbb{I}$ is a free distributive lattice and whose cofibration equations are unconstrained, the naive approach is unlikely to be practical. To factor an equation into join-prime elements, the two terms being equated are first normalized. So unlike for the two cubical type theories just mentioned, factoring an equation into a cofibration over join-prime equations has exponential run-time. Hence, the naive decision procedure has double-exponential run-time.

In this talk, we will present a method for efficiently reducing cofibration entailment problems to the problem of deciding Boolean tautology of DNF formulas. Recalling that normalization trivializes the latter problem, this implies a significant practical speed up for all the cubical type theories mentioned above. We will also show correctness with respect to cubical presheaf semantics.

References

[1] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. Syntax and models of cartesian cubical type theory. *Mathematical Structures in Computer Science*, 31(4):424–468, 2021.

[2] Ulrik Buchholtz and Edward Morehouse. Varieties of cubical sets. *Lecture Notes in Computer Science*, page 77–92, 2017.

[3] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom, 2016.

[4] Harry B. Hunt, Daniel J. Rosenkrantz, and Peter A. Bloniarz. On the computational complexity of algebra on lattices. *SIAM J. Comput.*, 16:129–148, 1987.

[5] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories, 2017.

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

[7] Matthew Z. Weaver and Daniel R. Licata. A constructive model of directed univalence in bicubical sets. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, page 915–928, New York, NY, USA, 2020. Association for Computing Machinery.

Description Operators in Partial Propositional Type Theory

Víctor Aranda[1]
vicarand@ucm.es
Manuel Martins[2]
martins@ua.pt

[1] Department of Logic and Theoretical Philosophy, Complutense University of Madrid, Spain.
[2] Department of Mathematics, University of Aveiro, Portugal.

A Type Theory containing $\{T, F\}$ as only basic type is Leśniewski's *Protothetics* [2]. Henkin [3] arrived at this formal system independently of Leśniewski, presenting a (complete) calculus for Propositional Types using only λ and identity as primitive symbols. The completeness proof lies in the fact that it is possible to have a *name* in the object language for every element of the hierarchy of types (this hierarchy is obtained by passing from any two sets $\mathcal{D}_\alpha$ and $\mathcal{D}_\beta$ to a set $\mathcal{D}_{\alpha\beta}$). Since the elements of the domains above $\{T, F\}$ —above $\mathcal{D}_t$— are functions, the addition of an *undefined value* to $\mathcal{D}_t$ naturally produces functions which are undefined when applied to certain arguments (i.e., functions for which we can compute that there is no result). Therefore, a 3-valued Propositional Type Theory seems to be a suitable vehicle for dealing with *partial functions*. A semantics for such a logic was given by Lepage [5] and Lapierre [4].

However, not every function from $\{T, F, \divideontimes\}$ to $\{T, F, \divideontimes\}$ should be considered as a partial function. The behavior of the undefined truth-value has to be compatible with any increase in information, that is, it must be *regular*. Lepage [5], who found inspiration in Kleene's undefined value, restricted the set of partial functions in $\mathcal{D}^{\divideontimes}_{tt}$ to those that are monotonic with respect to the following partial order (on the new basic type): $\divideontimes \leq \divideontimes$, $\divideontimes \leq T$, $\divideontimes \leq F$, $T \leq T$ and $F \leq F$. Monotonicity is imposed to any domain $\mathcal{D}_{\alpha\beta}$.

It is worth recalling at this point that the set of monotonic functions from $\mathcal{D}^{\divideontimes}_{t}$ to $\mathcal{D}^{\divideontimes}_{tt}$ has many interesting functions: in particular, *all* Kleene's strong connectives. In fact, the name for conjunction in Henkin's hierarchy is now the name for Kleene's strong conjunction (see [6, p. 33]), provided that identity is interpreted in a different manner (see [4, p. 533]). The names for the remaining Kleene's connectives involve conjunction and negation. Lepage's 3-valued logic also distinguishes between total and non-total objects, which are defined inductively (see [4, p. 527]), and includes a function symbol $\mathfrak{J}(A_\alpha)$ saying that "the interpretation of A_α is a total object".

Lepage [6] offers a sound axiomatic system for this 3-valued Propositional Type Theory, but its completeness is posed as an open question. The lack of proof is due to the "impossibility of having a canonical name in the object language for every partial function without modifying the theoretical framework in an essential way" (Lepage [6, p. 37]).

The aim of this paper is to define description operators for Lepage's system (the first step towards a Nameability Theorem) by means of suitable modifications and exploiting the semantics of Kleene's strong connectives. Firstly, we will explain why Henkin's description operator for type $\langle tt\rangle$ cannot be simply transferred to the domain $\mathcal{D}^{\divideontimes}_{tt}$ of the 3-valued hierarchy. Then, we will define an election function à la Henkin (see [3, p. 328]) of type $\langle \alpha t\rangle\alpha$, proving that such a function *is* in the 3-valued hierarchy, i.e., its monotonicity. We will also show that the denotation of the expression

$$\iota_{\langle tt\rangle t} \coloneqq (\lambda f_{tt}(\mathfrak{J}(f_{tt}) \to f_{tt} \equiv (\lambda x_t x_t)))$$

is the election function for $\mathcal{D}^{*}_{tt}$ (where $\rightarrow$ is Kleene's strong conditional). Finally, we will outline how to prove that, for any α, there is a closed expression $\iota_{\langle\alpha t\rangle\alpha}$ such that $(\iota_{\langle\alpha t\rangle\alpha})^d = \mathbf{t}^{(\alpha)}$ where $\mathbf{t}^{(\alpha)}$ is the election function for the arbitrary type α. To state this result, we have to (1) add a proper symbol U_α (of type α) to the set of primitive symbols of Propositional Type Theory and (2) build a "lambda hierarchy of undefinedness" whose interpretation is, for each level of the hierarchy, (3) the *least defined element* of the corresponding domain. The following definitions are introduced:

Definition 1 (Lambda hierarchy of undefinedness). We inductively define U_σ for any type σ:

1. For $\sigma = t$, $U_\sigma \coloneqq U_t$.
2. For $\sigma = \alpha\beta$, $U_\sigma \coloneqq \lambda x_\alpha U_\beta$.

Definition 2 (Least defined element). We inductively define $\mathbf{u}_\sigma$ for any type σ:

1. For $\sigma = t$, $\mathbf{u}_\sigma \coloneqq *$.
2. For $\sigma = \alpha\beta$, $\mathbf{u}_\sigma$ is the function of $\mathcal{D}_{\alpha\beta}$ such that, for any $\theta \in \mathcal{D}_\alpha$, $\mathbf{u}_{\alpha\beta}(\theta) = \mathbf{u}_\beta$.

We stipulate that the interpretation of any element U_σ of the lambda hierarchy (which found inspiration in [1]) is $\mathbf{u}_\sigma$. Hence, the undefined value, and any function of type $\alpha\beta$ sending every element of $\mathcal{D}^{*}_{\alpha}$ to $\mathbf{u}_\beta$, will finally have a name in the object language. From a purely philosophical point of view, the fact that we can designate functions that are completely undefined may be problematic. What kind of *object* is a function that is undefined for every element of its domain? Should we even call it "partial function"? We think of these functions as *extreme cases* that serve us as denotation for those expressions pointing at non-existent functions of the 3-valued hierarchy. Let us conclude by explaining this perspective in some detail.

Lepage [6, p. 37] explicitly rejected the possibility of including a name for the third truth-value in the object language, for the reason that "one immediate consequence would be that, even for partial total valuations, some expressions would remain undefined". However, we claim that, once $\{U_\alpha\}_{\alpha\in\mathsf{PT}}$ (where PT is the set of type symbols) has been added to the set of primitive symbols, and having defined the lambda hierarchy of undefinedness, it holds for any meaningful expression A_α that its interpretation is in $\mathcal{D}_\alpha$.

To sum up, the talk will present several syntactic and semantic modifications of the Partial Propositional Type Theory developed in [5], [4] and [6], which is nothing but a Kleene's logic. Although the most recent research on Henkin's axiomatic system does not focus on making it many-valued (see, among others, [7] and [8]), our goal is to show that these modifications allow us to define description operators even with a third truth-value in play.

References

[1] William M Farmer (1990): *A partial functions version of Church's simple theory of types. The Journal of Symbolic Logic* 55(3), pp. 1269–1291.

[2] Andrzej Grzegorczyk (1955): *The systems of Leśniewski in relation to contemporary logical research. Studia Logica* 3(1), pp. 77–95.

[3] Leon Henkin (1963): *A theory of propositional types. Fundamenta Mathematicae* 52, pp. 323–344.

[4] Serge Lapierre (1992): *A functional partial semantics for intensional logic. Notre Dame Journal of Formal Logic* 33(4), pp. 517–541.

[5] François Lepage (1992): *Partial functions in type theory. Notre Dame Journal of Formal Logic* 33(4), pp. 493–516.

[6] François Lepage (1995): *Partial propositional logic*. In: *Québec Studies in the Philosophy of Science*, Springer, pp. 23–39.

[7] Maria Manzano, Manuel Martins & Antonia Huertas (2014): *A semantics for equational hybrid propositional type theory*. *Bulletin of the Section of Logic* 43(3), p. 4.

[8] Maria Manzano, Manuel Martins & Antonia Huertas (2019): *Completeness in equational hybrid propositional type theory*. *Studia Logica* 107(6), pp. 1159–1198.

Exhaustive testing *of* property testers

Auke Booij

March 2022

Property testers only evaluate properties at a finite number of inputs, and so even a property that passed testing might still not hold for all inputs. We can also phrase properties *of* property testers. Perhaps surprisingly, this higher-order problem is decidable using Escardó's infinite search [2], subject to some caveats.

Property testing is used to test program correctness over a range of inputs [1]. In a highly simplified setting, the property we want to test is given to the property tester as a function. For example, we can phrase the property that 0 is a right identity of + as the Haskell function:

```haskell
prop_assoc :: Natural -> Bool
prop_assoc i = (i + 0 == i)
```

A property tester such as QuickCheck would then evaluate this function several times with different input values.

- If the function returned `True` every time, the property is considered satisfied.
- If a counterexample is found, then this is reported to the user.

We can make this more concrete by defining an abstract property tester as follows.

```haskell
type Property range = range -> Bool
data Result counterexample
  = Success
  | Failure counterexample
type Tester a = Property a -> Result a
```

With the above definitions, it is possible to phrase correctness criteria *of* such `Tester`s.

Here's one concrete example of a correctness criterion *of* property testers: if running a property tester results in a `Failure`, i.e. if it found a counterexample to a property, then the property better be refuted by that countexample. Below is the Haskell encoding of that.

```haskell
prop_failureActuallyFails :: Tester Natural -> Property (Property Natural)
prop_failureActuallyFails tester prop =
  case tester prop of
    Success -> True
    -- The property tester found a counterexample, so that better be one
    Failure n -> not (prop n)
```

Having arrived at this idealized view of property testing, we can state the central idea of this abstract: such properties *of* property testers are testable *exhaustively*.

In the talk, I'll also detail some other properties that can be tested exhaustively using this technique. The general, though perhaps unhelpful, answer is that we can test those properties that can be instantiated as a function `Tester Natural -> Property (Property Natural)`.

In other words, we can *decide*, for a given value `Tester Natural`, whether `prop_failureActuallyFails` is satisfied for *all* properties of *natural numbers*, i.e. for all inputs `Property Natural`. This is a straightforward application of the exhaustibility of Cantor space [2].

To be clear, this exhaustibility of Cantor space does not imply that we can enumerate all elements of the type `Natural -> Bool`, as this would certainly not be a finite list.

There are two caveats:

1. This only allows to decide correctness criteria of property testers instantiated at (a type isomorphic to) the natural numbers. However, in general, property testers may be polymorphic. What can we say for instantiations at other types?

   (a) We can consider other types X that make $X \to \mathsf{Bool}$ exhaustible: we can exhaustively test property testers instantiated at any such X. However, this is a strong requirement on X, and so only has limited applications.

   (b) Normally, we use parametricity [3] to generalize properties of polymorphic functions from one instantiation to another. However, this works only to a very limited extent, since most property testers only work for a restricted selection of range types (e.g. types for which elements can be generated), and so are not fully polymorphic functions. So the free theorems yielded by parametricity are limited, roughly in proportion to the restrictions on the range type of the property tester.

2. Normally, property testers use (pseudo)random generators to choose input values from the range. Such a pseudorandom property tester can be interpreted as a deterministic one by choosing a seed, and thus randomized property testers correspond to a *family* of deterministic property testers. Thus, while we could exhaustively test a property tester *with a fixed seed*, strictly speaking this proves nothing about the rest of the family.

This idea has been implemented for QuickCheck; no bugs were found. The presence of IO was not problematic in the case of QuickCheck, but in general it could be.

References

[1] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. Ed. by M. Odersky and P. Wadler. ACM, 2000, pp. 268–279. DOI: 10.1145/351240.351266.

[2] M. H. Escardó. "Exhaustible Sets in Higher-type Computation". In: *Log. Methods Comput. Sci.* 4.3 (2008). DOI: 10.2168/LMCS-4(3:3)2008.

[3] P. Wadler. "Theorems for Free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. 1989, pp. 347–359. DOI: 10.1145/99370.99404.

Extending Cubical Agda with Internal Parametricity*

Antoine Van Muylder[1], Andrea Vezzosi, Andreas Nuyts[1], and Dominique Devriese[1]

[1]KU Leuven, Belgium

Abstract. Internally parametric type theories are type systems augmented with additional primitives and typing rules allowing the user to prove parametricity statements within the system, without resorting to axioms. We implement such a type system by extending the cubical type theory of Cubical Agda [17] with parametricity primitives proposed by Cavallo and Harper [9]. To assess the implementation, we formalize a general parametricity theorem within the system, which entails a large spectrum of free theorems including a Church encoding for the circle and a straightforward parametric model for System F[1].

A type-polymorphic function is *parametric* if its type argument is merely used for typing, not for computing purposes. Such a parametric function necessarily applies the same algorithm irrespective of the type it is being used at. Reynolds' relational parametricity [15, 13] is a semantic account of this property for, e.g., terms of System F (a.k.a. the second-order polymorphic lambda calculus). Useful information can systematically be extracted by only looking at the type of a parametric function. These facts commonly known as "free theorems" [18] provide, for instance, a formal explanation as to why there are only two functions with type $\forall\alpha.\,\alpha \to \alpha \to \alpha$. Indeed, in a parametric model (where all denotations are parametric), only the first and second polymorphic projections qualify as valid, as they do not inspect the implementation of α.

Enforcing free theorems. Dependent type theory (DTT) has been proven to admit parametric models [16, 6, 12, 3]. Therefore, whenever a free theorem is needed, it can soundly be added as an axiom. In fact, evidence that a given closed term is parametric (w.r.t. a syntactic notion of relation) can even be obtained constructively: this is what *parametricity translations* [11, 1] achieve. However, parametricity is known to be logically independent from plain DTT [7] and this prevents the above meta-theoretical translations to be internalized. Hence, to obtain internal parametricity, novel principles must be added to plain DTT.

Internal parametricity for cubical type theory. Cavallo and Harper (CH) [9] extend cubical type theory [2] with parametricity primitives [5], in a style reminiscent of cubical type theory itself. Proofs of *relatedness* (versus equality) between $a_0, a_1 : A$ are built using functions $p : \mathsf{BI} \to A$ from an abstract *bridge interval* BI, satisfying $p(0) = a_0$, $p(1) = a_1$ definitionally. Such proofs are called *bridges* and written $\lambda^{\mathsf{BI}}\, r.\, p(r) : \mathsf{Bridge}_A\, a_0\, a_1$ (versus paths $\lambda^{\mathsf{I}}\, i.\, p(i) : \mathsf{Path}_A\, a_0\, a_1$). Contrary to path variables, the logic of bridge variables is sub-structural (*affine*): weakening and exchange hold, but not contraction. Concretely, one can eliminate a bridge $\Gamma_1, r : \mathsf{BI}, \Gamma_2 \vdash b : \mathsf{Bridge}_A\, a_0\, a_1$ at a bridge variable r only if r is *fresh* for b, meaning that every free variable appearing in b is in Γ_1 or is a bridge/path variable in Γ_2. This sub-structurality is crucial to formulate the inference rules of the extent and Gel primitives. The purpose of those primitives is to guarantee several *bridge commutation principles*: theorems explaining how the Bridge type former commutes with other type formers. The extent primitive and its rules provide commutation with Π. For non-dependent functions this reads

*Van Muylder/Nuyts hold a PhD/Postdoctoral fellowship of the Research Foundation – Flanders (FWO).

[1]Files, instructions, comparison with other systems: https://github.com/antoinevanmuylder/bridgy-lib.

$\left(\Pi_{(a_0,a_1:A)(\bar{a}:\mathsf{Bridge}\ a_0\ a_1)}\mathsf{Bridge}_B(f_0a_0)(f_1a_1)\right) \simeq \mathsf{Bridge}_{A\to B}\ f_0\ f_1$. The principle is analogous to function extensionality and asserts that functions are related if they map related inputs to related outputs. The Gel primitive and its rules, together with univalence, prove commutation with the universe: $(A_0 \to A_1 \to \mathsf{Type}) \simeq \mathsf{Bridge}_{\mathsf{Type}}\ A_0\ A_1$. This is analogous to univalence and called relativity by CH. Commutation principles make bridges (and paths) behave as structured relations (and isomorphisms, resp.). This is most blatant for types of algebraic structures. Consider the type of "magmas" $\mathsf{Mag} = \Sigma_{M:\mathsf{Type}}\, M \times M \to M$. Commutation with $\Sigma, \mathsf{Type}, \times, \to$ grants a characterization of $\mathsf{Bridge}_{\mathsf{Mag}}\, M_0\, M_1$ (and Path, resp.) as the type of relations (and isomorphisms, resp.) compatible with the binary functions from M_0, M_1. Such structured relations are also known as *logical relations* [10].

Contributions. We implement CH's internally parametric type theory [9] on top of the cubical type theory [8] underlying the Cubical Agda [17] proof assistant. As discussed above, we must be able to generate freshness constraints for bridge variables during typechecking. Our implementation hence reuses the existing affine variable infrastructure of Guarded Cubical Agda [14]. Interestingly and unprecedented in Agda, the extent_β computational rule fires only if a certain argument M satisfies a specific freshness condition. As this condition is not reflected by β-reduction, the rule has to operate on the normal form of M in the worst case scenario. Our current implementation of extent_β is sound but not complete because of this peculiar behaviour. Implementation of Kan operations for $\mathsf{Bridge}, \mathsf{Gel}$ is work in progress as well.

Our long term goals include assessing the precise expressivity of CH's internal parametricity, connecting it to existing alternate formulations (unary, Kripke, etc.) and evaluating its usefulness in practical applications. For now, we have already formalised a (binary) parametricity statement from which a wide range of free theorems ensue. A *native reflexive graph* is by definition a type of vertices G equipped, for any $g_0, g_1 : G$ with a type of edges $G\{g_0, g_1\}$ and an equivalence $\eta^G : G\{g_0, g_1\} \simeq \mathsf{Bridge}_G\, g_0\, g_1$. The type of native reflexive graphs is of course equivalent to Type, but η^G can contain non-trivial information. For instance, Type equipped with relations $A_0 \to A_1 \to \mathsf{Type}$ as edges is native exactly thanks to relativity, and formalizing the relativity theorem was non-trivial. Similarly, a *native relator* F between native reflexive graphs $G, H : \mathsf{Type}$ acts both on vertices $F_{\mathrm{vrt}} : G \to H$ and on edges $F_{\mathrm{edge}}^{g_0,g_1} : G\{g_0, g_1\} \to H\{Fg_0, Fg_1\}$ and the latter action must satisfy $\mathsf{Path}_{\dots}(F_{\mathrm{bdg}} \circ \eta^G)(\eta^H \circ F_{\mathrm{edge}})$ where $F_{\mathrm{bdg}} = (\lambda q.\ \lambda^{\mathsf{BI}} x.\ F_{\mathrm{vrt}}(qx))$. Parametricity now reads as follows: for any native relator $F : G \to \mathsf{Type}$ and any function $f : \Pi_{x:G}\, Fx$, inputs related by an edge $e : G\{g_0, g_1\}$ result in a proof $(\mathsf{param} : F_{\mathrm{edge}}\, e\, (fg_0)\, (fg_1))$. Bridge commutation principles ensure that native relators abound. For instance the arrow relator $\mathsf{Type} \times \mathsf{Type} \to \mathsf{Type} : A, B \mapsto A \to B$ is native, and using the above param constant it is easy to show that $\left(\Pi_{(X:\mathsf{Type})} X \to X \to X\right) \simeq \mathsf{Bool}$

Less standard is the following Church encoding for the circle [4]: $\left(\Pi_{X_*:\mathsf{Type}_*} \Omega(X_*) \to X\right) \simeq S^1$, where $\mathsf{Type}_* = \Sigma_{X:\mathsf{Type}} X$ and $X_* = (X, x_0)$ and $\Omega(X_*) = \mathsf{Path}_X\, x_0\, x_0$. The above param constant provides a direct proof, granted that $\lambda X_*.\ \Omega(X_*) \to X$ is a native relator. The formal proof of Ω nativeness is ongoing.

Finally we connect Reynolds' statement to ours and describe a shallow embedding of System F that can serve as a parametric model. We wrongly assume $\mathsf{Type} : \mathsf{Type}$ to comply with System F impredicativity and simplify matters. Semantic open types $\alpha_1, ..., \alpha_n : * \models \tau : *$ are defined as native relators $\mathsf{Type} \times ... \times \mathsf{Type} \to \mathsf{Type}$. There is a semantic arrow type given by the above arrow relator, and a semantic $\forall$ type as well. Semantic open terms $(\alpha_1, ..., \alpha_n : *) \mid (x_1 : \tau_1, ..., x_m : \tau_m) \models t : \tau$ are functions $\Pi_{\theta:\mathsf{Type}^n}\, \Pi_j \tau_j(\theta) \to \tau(\theta)$. Once again, as $\lambda\theta.\ \Pi_j \tau_j(\theta) \to \tau(\theta)$ is a native relator, semantic terms are proven parametric thanks to param.

References

[1] Abhishek Anand and Greg Morrisett. Revisiting Parametricity: Inductives and Uniformity of Propositions. *arXiv:1705.01163 [cs]*, July 2017.

[2] Carlo Angiuli, Robert Harper, et al. Cartesian cubical computational type theory: Constructive reasoning with paths and equalities. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[3] Robert Atkey, Neil Ghani, and Patricia Johann. A Relationally Parametric Model of Dependent Type Theory. In *Principles of Programming Languages*, pages 503–515. ACM, 2014.

[4] Steve Awodey, Jonas Frey, and Sam Speight. Impredicative encodings of (higher) inductive types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 76–85, 2018.

[5] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electronic Notes in Theoretical Computer Science*, 319:67–82, 2015.

[6] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 345–356, 2010.

[7] Auke B Booij, Martín H Escardó, Peter LeFanu Lumsdaine, and Michael Shulman. Parametricity, automorphisms of the universe, and excluded middle. In *22nd International Conference on Types for Proofs and Programs*, 2018.

[8] 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.

[9] Robert Harper and Evan Cavallo. Internal parametricity for cubical type theory. *Logical Methods in Computer Science*, 17, 2021.

[10] Claudio Hermida, Uday S Reddy, and Edmund P Robinson. Logical relations and parametricity–a Reynolds programme for category theory and programming languages. *Electronic Notes in Theoretical Computer Science*, 303:149–180, 2014.

[11] Chantal Keller and Marc Lasson. Parametricity in an impredicative sort. In *Computer Science Logic (CSL'12)-26th International Workshop/21st Annual Conference of the EACSL*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[12] Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, pages 432–451, 2013.

[13] QingMing Ma and John C Reynolds. Types, abstraction, and parametric polymorphism, part 2. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 1–40. Springer, 1991.

[14] 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.

[15] John C Reynolds. Types, abstraction and parametric polymorphism. In *IFIP congress*, volume 83, 1983.

[16] Izumi Takeuti. The theory of parametricity in lambda cube. Technical report 1217, Kyoto University, 2001.

[17] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31, 2021.

[18] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

Extending truth table natural deduction to predicate logic

Herman Geuvers[1] and Tonny Hurkens

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

We extend *truth table natural deduction* (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 [1, 2]) to predicate logic. We have two variations on the rules, one is a complete natural deduction calculus for classical logic, while the other is a complete natural deduction calculus for constructive logic.

Monotonic and non-monotonic quantifiers A main observation from earlier work is that for monotonic connectives (that is, connectives that come from a truth table that corresponds to a monotonic function from $\{0, \ldots, 1\}^n$ to $\{0, 1\}$, as induced by the $0 \leq 1$ ordering), the constructive and classical rules are equivalent [4]. For non-monotonic connectives: one classical non-monotonic connective makes the whole calculus classical. Naosuke Matsuda and Kento Takagi [3] extend this result by adding $\forall$ and $\exists$. For this, the Kripke models are restricted to those in which the (non-empty) domain $D(w)$ of individuals at w is a constant (instead of monotonic) function of w.

In this paper, we observe that quantifiers can themselves be monotonic or non-monotonic, and that they have constructive and classical deduction rules, which are equivalent for the monotonic connectives, but different for non-monotonic ones. For standard predicate logic, the existential quantifier $\exists$ is monotonic, while the universal quantifier $\forall$ is non-monotonic. This can be observed when thinking about Kripke models. If $\exists x.\varphi$ holds in a world w, it automatically holds in all worlds $w' \geq w$, because the domain element $d \in D(w)$ for which φ holds is still available in w' (or put differently: the truth of $\exists x.\varphi$ in a world can be defined locally). On the contrary, to know whether $\forall x.\varphi$ holds in a world w, it doesn't suffice to know that φ holds in w for all $d \in D(w)$. We also have to know this for all $d' \in D(w')$ for all worlds $w' \geq w$ (or put differently: the validity of $\forall x.\varphi$ in a world can only be define by inspecting all higher worlds).

There are various examples that show the difference between classical and constructive predicate calculus. One of them is known as the Drinker's Principle: $\exists y.(Dy \to \forall x.Dx)$. This can be proven by using a case distinction on $\forall x.Dx$ or $\exists x.\neg Dx$, and if one wants to limit the language to the connectives available in the formula, by using classical rules for implication. But it can also be shown using the classical rules for $\forall$ (to be given below) and constructive rules for implication. Another example is $\forall x.(Ax \vee Ex) \vdash \forall x.Ax \vee \exists x.Ex$, which we prove using our classical $\forall$-rules. (Note that if one uses the standard rules for $\exists$, $\forall$ and $\vee$, one has to use classical negation as an auxiliary connective to prove this formula!)

We present our derivation rules in a condensed form, leaving out auxiliary assumptions and conclusions. The rules are instantiated to derive sequents of the form $\Gamma; A \vdash \varphi$, where Γ is a finite set of formulas, φ is a single formula and A is a set of constants. We only give the rules for $\forall$ and $\exists$, as the rules for propositional connectives are standard (and in our format they can be found in [1, 2]).

First-order calculus Let L be a first-order language. For each finite set A of new individual constants we extend L to language $L[A]$. A *first-order sequent* $\Gamma; A \vdash \varphi$ consists of such a set A, a finite set of closed formulas of $L[A]$, Γ and a closed formula φ of $L[A]$. We recursively extend

L with new individual constants $a = a_\psi$ related to closed formulas ψ of the form $\exists x.\varphi$ or $\forall x.\varphi$. Constants added before may occur in ψ. In a classical model with (non-empty) domain D, the interpretation $d \in D$ of the constant a related to such a formula ψ should be such that the closed formulas ψ and $\varphi[a/x]$ are both true or both false in that model.

Inference rules are now operations on first-order sequents. The sequents used in the specification are differences of first-order sequents: only the parts that differ are written. The start rule (or "axiom") is extended as follows:

$$\boxed{\frac{}{\varphi; A \vdash \varphi}\, S}$$

We propose the following inference rules for the quantifiers. Rules indicated with a * are classical; similarly to the propositional connectives, only the introduction rules are different classically. For $\exists$, the classical and constructive introduction rule are equivalent.

$$\boxed{\frac{\vdash \exists x.\varphi \quad ; a \vdash}{\vdash}\, E\exists} \qquad \boxed{\frac{\vdash \exists x.\varphi \quad \varphi[a/x]; \vdash}{\vdash}\, E\exists}$$

$$\boxed{\frac{\vdash \varphi[t/x]}{\vdash \exists x.\varphi}\, I\exists} \qquad \boxed{\frac{\exists x.\varphi; \vdash \quad \vdash \varphi[t/x]}{\vdash}\, I^*\exists}$$

$$\boxed{\frac{\vdash \forall x.\varphi}{\vdash \varphi[t/x]}\, E\forall} \qquad \boxed{\frac{\vdash \forall x.\varphi \quad \varphi[t/x]; \vdash}{\vdash}\, E\forall}$$

$$\boxed{\frac{; a \vdash}{\vdash \forall x.\varphi}\, I\forall} \qquad \boxed{\frac{\vdash \varphi[a/x]}{\vdash \forall x.\varphi}\, I\forall}$$

$$\boxed{\frac{\forall x.\varphi; \vdash \quad ; a \vdash}{\vdash}\, I^*\forall} \qquad \boxed{\frac{\forall x.\varphi; \vdash \quad \vdash \varphi[a/x]}{\vdash}\, I^*\forall}$$

For the left $E\exists$ rule (and similarly for the left $I\forall$ rules), when we instantiate this rule to a concrete sequent, the a should not occur anywhere else in the formulas, otherwise the sequents are not well-formed. So a concrete application of the left $E\exists$ rule looks like this:

$$\frac{\Gamma_1; A_1 \vdash \exists x.\varphi \quad \Gamma_2; A_2, a \vdash \chi}{\Gamma_1, \Gamma_2; A_1, A_2 \vdash \chi}\, E\exists$$

where a does not occur in Γ_2, χ, otherwise $\Gamma_1, \Gamma_2; A_1, A_2 \vdash \chi$ is not a first-order sequent.

Note that application of a constructive $\forall$-introduction rule can be seen as composition: application of the corresponding classical introduction rule in which the first premise is the result of a start rule application of the form $\forall x.\varphi; A \vdash \forall x.\varphi$.

We give one example of a natural deduction of a classically valid sequent which has the subformula property. We derive $\varphi; \vdash \chi$ where $\varphi = \forall x.(Ax \vee Ex)$ and $\chi = \forall x.Ax \vee \exists x.Ex$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x.Ax; \vdash \forall x.Ax}\, S}{\forall x.Ax; \vdash \chi}\, I\vee \qquad \dfrac{\dfrac{\dfrac{}{\varphi; a \vdash \forall x.(Ax \vee Ex)}\, S}{\varphi; a \vdash Aa \vee Ea}\, E\forall \qquad \dfrac{\dfrac{\dfrac{}{Aa; a \vdash Aa}\, S}{Aa; a \vdash \forall x.Ax}\, I\forall \qquad \dfrac{\dfrac{}{Ea; a \vdash Ea}\, S}{Ea; a \vdash \exists x.Ex}\, I\exists}{Aa; a \vdash \chi}\, I\vee \qquad \dfrac{}{Ea; a \vdash \chi}\, I\vee}{\varphi; a \vdash \chi}\, E\vee}{\varphi; \vdash \chi}\, I^*\forall$$

References

[1] 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.

[2] 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.

[3] Naosuke Matsuda and Kento Takagi. Effect of the choice of connectives on the relation between the logic of constant domains and classical predicate logic. *CoRR*, arXiv:2107.03972, 2021. URL: https://doi.org/10.48550/arXiv.2107.03972, arXiv:2107.03972.

[4] I. van der Giessen. Natural deduction derived from truth tables, master thesis mathematics, radboud university nijmegen, July 2018. URL: https://www.ru.nl/math/@1060423/master-theses-per-year/.

Flexible presentations of graded monads

Shin-ya Katsumata[1], Dylan McDermott[2], Tarmo Uustalu[23], and Nicolas Wu[4]

[1] National Institute of Informatics, Japan s-katsumata@nii.ac.jp
[2] Reykjavik University, Iceland dylanm@ru.is,tarmo@ru.is
[3] Tallinn University of Technology, Estonia
[4] Imperial College London, UK n.wu@imperial.ac.uk

Consider a language in which we can express backtracking computations using an operation `or` for nondeterministic choice, and an operation `cut` for pruning any remaining choices. Let t be the computation $\texttt{or}(\texttt{return}\ 17, \texttt{cut})$, which offers only 17 as a possible result, and prunes the rest of the search space. The computation $\texttt{or}(t, \texttt{return}\ 42)$ is equivalent to t, and more generally, the equation $\texttt{or}(x, y) \approx x$ is valid whenever we know that x definitely `cut`s. We may seek to analyse computations statically to determine whether they `cut`, and whether we can therefore simplify a program using $\texttt{or}(x, y) \approx x$. One approach to doing this is through *grading*. We assign a grade $\bot$ to each computation we know will `cut`, and propagate this information throughout the program (other computations get other grades). This approach has a well-established semantics using *graded monads* [9, 5, 2]. There is a graded monad Cut that models our backtracking example; it is similar to Piróg and Staton's non-graded monad [7]. Piróg and Staton show that their monad has a *presentation* in terms of operations for nondeterministic choice and cut. We may expect there to be a similar presentation of Cut, using the existing notions of graded presentation [9, 6, 1, 3], which we call *rigidly graded presentations*. However, rigidly graded presentations have a deficiency: they only allow operations to be applied when all arguments have the same grade. Above t has grade $\bot$ because one argument to `cut` has grade $\bot$, but the other does not. A rigidly graded presentation would assign *some* grade to t, by overapproximating, but not $\bot$, so the analysis would be imprecise. This is a problem in other applications, such as: mutable state graded by relations (relating initial states to final states); stack-based computations graded by bounds on the change in stack height; and nondeterministic computations graded by upper bounds on the number of options that are chosen from.

While rigidly graded presentations are motivated by their theory (which includes a correspondence with a class of graded monads, analogous to the classical monad–algebraic theory correspondence), they are unsuitable when it comes to applications. We introduce a more general notion of *flexibly graded* presentation that does not suffer from the same issue.

Grading We recall the notion of graded monad (on **Set**). The *grades* are elements of an ordered monoid $(|\mathbb{E}|, \leq, 1, \cdot)$. A grade $e \in |\mathbb{E}|$ abstractly quantifies the effect of a computation; the order $\leq$ provides overapproximation of grades, the unit 1 is the grade of a trivial computation, and the multiplication $\cdot$ provides the grade of a sequence of two computations. For the backtracking example above the poset $(|\mathbb{E}|, \leq)$ is $\{\bot \leq 1 \leq \top\}$, where $\bot$ means 'definitely cuts', the unit grade 1 means 'definitely either cuts or produces at least one value', and $\top$ imposes no restrictions. Multiplication is given by $\bot \cdot e = \bot$, $1 \cdot e = e$ and $\top \cdot e = \top$.

A *graded set* Y is a family of sets Ye, together with a *coercion* function $(e{\leq}e')^* : Ye \to Ye'$ for each $e \leq e'$, satisfying two equational conditions. A *graded monad* R consists of a graded set RX and *unit* function $\eta_X : X \to RX1$ for each set X, and a *Kleisli extension* operation that maps functions $f : X \to RYe$ and grades d to functions $f^\dagger_d : RXd \to RY(d \cdot e)$, satisfying some conditions. For Cut, computations over X of grade e are elements of the following set $\mathrm{Cut}Xe$, where c indicates whether the computation cuts ($\bot$ for 'cuts', $\top$ for 'does not cut').

$$\mathrm{Cut}Xe = \{(\vec{x}, c) \in \mathrm{List}X \times \{\bot, \top\} \mid (e = \bot \Rightarrow c = \bot) \wedge (e = 1 \Rightarrow c = \bot \vee \vec{x} \neq [])\}$$

Flexibly graded presentations In general, a *presentation* (Σ, E) consists of a *signature* Σ, specifying the *operations* and inducing a notion of *term*, and a set E of *equational axioms*, inducing an *equational theory*.

A *flexibly graded signature* Σ consists of a set $\Sigma(\vec{d'}; d)$ of $(\vec{d'}; d)$-ary operations for each list of grades $\vec{d'}$ and grade d. (*Rigidly graded signatures* correspond to the special case in which every operation has $\vec{d'} = [1, \ldots, 1]$.) The terms over Σ are generated by the following rules for variables, coercions, and application of operations $\mathrm{op} \in \Sigma(\vec{d'}; d)$, where $\Gamma = x_1 : d'_1, \ldots, x_m : d'_m$.

$$\frac{1 \leq i \leq m}{\Gamma \vdash x_i : d'_i} \qquad \frac{\Gamma \vdash t : e \quad e \leq e'}{\Gamma \vdash (e \leq e')^* t : e'} \qquad \frac{\Gamma \vdash u_1 : d'_1 \cdot e \quad \cdots \quad \Gamma \vdash u_n : d'_n \cdot e}{\Gamma \vdash \mathrm{op}(e; u_1, \ldots, u_n) : d \cdot e}$$

The grade e in the op rule has a crucial role: it is there precisely because of the grade e in the Kleisli extension above. Unlike in a rigidly graded presentation, variables can have different grades d'_i. In a *flexibly graded presentation* (Σ, E), an equational axiom in E is a pair (t, u) of terms of some grade e in some context Γ. These axioms induce a notion of equality $\Gamma \vdash t \approx u : e$. For the backtracking example, we have a flexibly graded version of Piróg and Staton's non-graded presentation [7]. The signature has operations cut, fail, or_{d_1,d_2}, giving rise to the following rules for constructing terms (where $\sqcap$ denotes meet).

$$\frac{}{\Gamma \vdash \mathsf{cut}(e;\ ) : \bot} \qquad \frac{}{\Gamma \vdash \mathsf{fail}(e;\ ) : \top} \qquad \frac{\Gamma \vdash u_1 : d_1 \cdot e \qquad \Gamma \vdash u_2 : d_2 \cdot e}{\Gamma \vdash \mathsf{or}_{d_1,d_2}(e; u_1, u_2) : (d_1 \sqcap d_2) \cdot e}$$

One of the axioms (we omit the rest) is $x : \bot, y : 1 \vdash \mathsf{or}_{\bot,1}(1; x, y) \approx x : \bot$, which is the example we use in the introduction. This can be applied only when x has grade $\bot$; such a restriction on the grade of a variable is not possible in a rigidly graded presentation.

Semantics In classical universal algebra each presentation gives rise to a notion of *algebra* (a.k.a. *model*), consisting of a set with interpretations for the operations, validating the equations. The equational theory is sound and complete w.r.t. this notion of model. If (Σ, E) is a flexibly graded presentation, a Σ-*algebra* is a graded set A equipped with a natural transformation $[\![\mathrm{op}]\!] : \prod_i A(d'_i \cdot -) \Rightarrow A(d \cdot -)$ for each $\mathrm{op} \in \Sigma(\vec{d'}, d)$. These extend to interpretations $[\![t]\!] : \prod_i A(d'_i \cdot -) \Rightarrow A(d \cdot -)$ of terms $x_1 : d'_1, \ldots, x_n : d'_n \vdash t : d$. A Σ-algebra is a (Σ, E)-*algebra* when $[\![t]\!] = [\![u]\!]$ for each axiom (t, u). The equational logic is sound and complete: an equation $\Gamma \vdash t \approx u : e$ is derivable exactly when $[\![t]\!] = [\![u]\!]$ in every (Σ, E)-algebra.

Presenting graded monads In the classical correspondence between presentations and monads, the monad $\mathsf{T}^{(\Sigma,E)}$ induced by a presentation is completely determined by the fact that $\mathsf{T}^{(\Sigma,E)}$-algebras are equivalently (Σ, E)-algebras. For flexibly graded presentations the situation is more complex. In general, there is no graded monad whose algebras are (Σ, E)-algebras, and we do not get a *correspondence* with graded monads. However, every flexibly graded presentation does induce a canonical graded monad $\mathsf{R}^{(\Sigma,E)}$. Every (Σ, E)-algebra induces an $\mathsf{R}^{(\Sigma,E)}$-algebra, and $\mathsf{R}^{(\Sigma,E)}$ is in some sense the universal graded monad with this property (we omit the precise statement). Moreover, *free* $\mathsf{R}^{(\Sigma,E)}$-algebras form (Σ, E)-algebras, so in particular the graded sets $R^{(\Sigma,E)}X$ admit interpretations of the operations of Σ. These interpretations form *flexibly graded algebraic operations* for $\mathsf{R}^{(\Sigma,E)}$ (which are analogous to algebraic operations for non-graded monads [8]). In this sense, (Σ, E) does indeed present a graded monad $\mathsf{R}^{(\Sigma,E)}$.

The proof of this involves a notion of *flexibly graded monad*, introduced in [4]. There is an algebra-preserving correspondence between flexibly graded presentations and flexibly graded monads that preserve *conical sifted colimits*, and every flexibly graded monad induces a canonical (rigidly) graded monad [4, Section 5]. The latter is $\mathsf{R}^{(\Sigma,E)}$ if we start with (Σ, E). Moreover, every graded monad R that preserves sifted colimits has a flexibly graded presentation.

References

[1] Ulrich Dorsch, Stefan Milius, and Lutz Schröder. Graded monads and graded logics for the linear time–branching time spectrum. In Wan Fokkink and Rob van Glabbeek, editors, *30th Int. Conf. on Concurrency Theory, CONCUR 2019*, volume 140 of *Leibniz Int. Proc. in Informatics*, pages 36:1–36:16. Dagstuhl Publishing, Saarbrücken/Wadern, 2019.

[2] Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proc. of 41st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 633–645. ACM Press, New York, 2014.

[3] Satoshi Kura. Graded algebraic theories. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures: 23rd Int. Conf., FOSSACS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*, volume 12077 of *Lect. Notes in Comput. Sci.*, pages 401–421. Springer, Cham, 2020.

[4] Dylan McDermott and Tarmo Uustalu. Flexibly graded monads and graded algebras. Manuscript, available at `https://dylanm.org/drafts/flexibly-graded-monads.pdf`, 2022.

[5] Paul-André Melliès. Parametric monads and enriched adjunctions. Manuscript, 2012.

[6] Stefan Milius, Dirk Pattinson, and Lutz Schröder. Generic trace semantics and graded monads. In Lawrence S. Moss and Paweł Sobociński, editors, *6th Conf. on Algebra and Coalgebra in Computer Science, CALCO 2015*, volume 35 of *Leibniz Int. Proceedings in Informatics*, pages 253–269. Dagstuhl Publishing, Saarbrücken/Wadern, 2015.

[7] Maciej Piróg and Sam Staton. Backtracking with cut via a distributive law and left-zero monoids. *J. Funct. Program.*, 27, 2017.

[8] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Appl. Categ. Struct.*, 11:69–94, 2003.

[9] A.L. Smirnov. Graded monads and rings of polynomials. *J. Math. Sci.*, 151(3):3032–3051, 2008.

Forwarders as Process Compatibility, Logically

Marco Carbone[1], Sonia Marin[2], and Carsten Schürmann[1]

[1] Computer Science Department, IT University of Copenhagen, Denmark
carbonem | carsten@itu.dk
[2] School of Computer Science, University of Birmingham, United Kingdom
s.marin@bham.ac.uk

Session types, originally proposed by Honda et al. [8], are type annotations that ascribe protocols to processes in a concurrent system and determine how they behave when communicating together. *Binary session types* found a logical justification in *linear logic*, identified by Caires and Pfenning [2] and later by Wadler [13, 14], which establishes the following correspondences: linear logic propositions *as* session types, proofs *as* processes, cut reductions in linear logic *as* reductions in the operational semantics, and duality *as* a notion of compatibility ensuring that two processes communications match.

The situation is not as direct for *multiparty session types* [9, 10], which generalize binary session types to protocols with more than two participants. One of the central observations is that compatibility requires a stronger property than duality, still ensuring that all messages sent by any participating party will eventually be collected by another. In [6], Deniélou and Yoshida first proposed the notion of *multiparty compatibility* as a "new extension to multiparty interactions of the duality condition for binary session type".

In this work, we carve out a fragment of classical linear logic (CLL) that adequately captures the notion of multiparty compatibility. We observe moreover that the processes corresponding to proofs in this fragment act as *forwarders* of messages between participants, generalizing existing proposals, such coherence [5], arbiters [3], and medium processes [1]. Our main result is that the multiparty compatibility of processes can be formalized by the means of a forwarder, and conversely, that any set of processes modelled by a forwarder is compatible.

Processes and types. The language of *processes* we use to represent communicating entities is a standard variant of the π-calculus [12] used in the context of session types: $P, Q ::= x \leftrightarrow y$ (link) $\mid x().P$ (wait) $\mid x[]$ (close) $\mid x(y).P$ (input) $\mid x[y \triangleright P].Q$ (output).

Types, taken to be propositions of CLL, denote the way an endpoint x (a channel's end) must be used at runtime. In this work, we must annotate each operator with another endpoint u (or a list of endpoints $\tilde{u}$ in the case of *gathering* of communications) to make explicit the destination of messages. Therefore, the syntax of types is defined as $A, B ::= a \mid a^{\perp} \mid \perp^{u} \mid \mathbf{1}^{\tilde{u}} \mid (A \parr^{u} B) \mid (A \otimes^{\tilde{u}} B)$, where a denotes atoms.

Multiparty compatibility. Multiparty compatibility [6, 11, 7] is a semantic notion that uses types as abstractions of the processes behaviors and simulates their execution.

In order to give a semantics to types, we introduce FIFO queues defined as $\Psi ::= \epsilon \mid A \cdot \Psi \mid * \cdot \Psi$, capturing messages that are waiting to be delivered. A *message* can be a proposition A or a session termination symbol $*$. Every pair of endpoints (x, y) has an associated queue containing messages in transit from endpoint x to endpoint y. A *queue environment* σ is a mapping from pairs of endpoints to queues: $\sigma : (x, y) \mapsto \Psi$. σ_ϵ denotes the queue environment with only empty queues while $\sigma[(x, y) \mapsto \Psi]$ denotes a new environment where the entry for (x, y) has been updated to Ψ.

We define the *type-context semantics* for context $\Delta = x_1 : B_1, \ldots, x_k : B_k$ given environment σ as the minimum relation on $\Delta \bullet \sigma$ following rules in Fig. 1. Let $\alpha_1, \ldots, \alpha_n$ denote a *path* for context Δ if $\Delta \bullet \sigma \xrightarrow{\alpha_1} \Delta_1 \bullet \sigma_1 \ldots \xrightarrow{\alpha_n} \Delta_n \bullet \sigma_n$. This path is *maximal* if there is no

$$
\begin{array}{lcl}
x : a^{\perp}, y : a \bullet \sigma_{\epsilon} & \xrightarrow{x \leftrightarrow y} & \varnothing \bullet \sigma_{\epsilon} \\
x : \mathbf{1}^{\tilde{y}} \bullet \ \sigma_{\epsilon}[\{(y_i, x) \mapsto *\}_i] & \xrightarrow{\tilde{y}\mathbf{1}x} & \varnothing \bullet \sigma_{\epsilon} \\
\Delta, x : \perp^{y} \bullet \sigma[(x, y) \mapsto \Psi] & \xrightarrow{x \perp y} & \Delta \bullet \sigma[(x, y) \mapsto \Psi \cdot *] \\
\Delta, x : A \parr^{y} B \bullet \sigma[(x, y) \mapsto \Psi] & \xrightarrow{x \parr y} & \Delta, x : B \bullet \sigma[(x, y) \mapsto \Psi \cdot A] \\
\Delta, x : A \otimes^{\tilde{y}} B \bullet \sigma[\{(y_i, x) \mapsto A_i \cdot \Psi_i\}_i] & \xrightarrow{\tilde{y} \otimes x[A, \{A_i\}_i]} & \Delta, x : B \bullet \sigma[\{(y_i, x) \mapsto \Psi_i\}_i]
\end{array}
$$

Figure 1: Type-Context Semantics

$$
\frac{}{x \leftrightarrow y \Vdash x : a^{\perp}, y : a} \ \text{Ax} \qquad \frac{P \Vdash \Gamma, [\![\Psi]\!][^{u} *]x : \cdot}{x().P \Vdash \Gamma, [\![\Psi]\!]x : \perp^{u}} \ \perp \qquad \frac{}{x[] \Vdash \{[^{x} *]u_i : \cdot\}_i, x : \mathbf{1}^{\tilde{u}}} \ \mathbf{1} \ (\tilde{u} \neq \varnothing)
$$

$$
\frac{P \Vdash \Gamma, [\![\Psi]\!][^{u} y : A]x : B}{x(y).P \Vdash \Gamma, [\![\Psi]\!]x : A \parr^{u} B} \ \parr \qquad \frac{P \Vdash \{y_i : A_i\}_i, y : A \quad Q \Vdash \Gamma, \{[\![\Psi_i]\!]u_i : A_i\}_i, [\![\Psi]\!]x : B}{x[y \triangleright P].Q \Vdash \Gamma, \{[^{x} y_i : A_i][\![\Psi_i]\!]u_i : C_i\}_i, [\![\Psi]\!]x : A \otimes^{\tilde{u}} B} \ \otimes \ (\tilde{u} \neq \varnothing)
$$

Figure 2: Proof System for Forwarders

$\Delta_{n+1}, \sigma_{n+1}, \alpha_{n+1}$ such that $\Delta_n \bullet \sigma_n \xrightarrow{\alpha_{n+1}} \Delta_{n+1} \bullet \sigma_{n+1}$ and it is *live* if $\Delta_n = \varnothing$ and $\sigma_n = \sigma_{\epsilon}$, meaning that it progresses without reaching an error.

Definition 1. *A context Δ is* multiparty compatible *if all maximal paths $\alpha_1, \ldots, \alpha_n$ for Δ are live and such that, if $\alpha_p = \tilde{y} \otimes x[A, \{A_i\}_i]$, then $x : A^{\perp}, \{y_i : A_i^{\perp}\}_i$ is also multiparty compatible.*

Forwarders. To capture multiparty compatibility, forwarders are designed as a subclass of derivations in classical linear logic that satisfies three conditions: i) anything received must be forwarded, ii) anything that is forwarded must have been previously received, and iii) the order of messages between two endpoints must be preserved.

In order to enforce these properties, the logic of forwarders is defined on judgements of the form $P \Vdash \Gamma$ where P is a process and $\Gamma ::= \varnothing \mid \Gamma, [\![\Psi]\!]x : B \mid \Gamma, [\![\Psi]\!]x : \cdot$ is an extended type context which associates to each endpoint x a priority queue $[\![\Psi]\!]$ consisting of any yet to process messages originating from x.

Formally, $[\![\Psi]\!] ::= \epsilon \mid [^{u} *][\![\Psi]\!] \mid [^{u} y : A][\![\Psi]\!]$ where $[^{u} y : A]$ expresses that a new y of type A has been received and will need to be forwarded later to endpoint u and, similarly, $[^{u} *]$ indicates that a request for closing a session has been received and must be forwarded to u.

Forwarders behave asynchronously: the order of messages needing to be forwarded to *independent* endpoints is irrelevant unless they originate at the same x. This follows the idea of having a queue for every ordered pair of endpoints in the type-context semantics above.

Definition 2. *A process P is a* forwarder *for Δ if the judgement $P \Vdash \Delta$ is derivable using the rules reported in Fig. 2.*

Correspondence. We can conclude this abstract with the statement of the theorem that forwarders characterize multiparty compatibility.

Theorem 3. *Δ is multiparty compatible iff there exists a forwarder for Δ, i.e., a process P such that $P \Vdash \Delta$.*

The proof, as well as more details on forwarders including additives for branching, exponentials to model servers and clients, internal cut-elimination for forwarder composition, and applications to multiparty communication *as* multi-cut elimination, can be found in [4].

References

[1] Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 74–95. Springer, 2016.

[2] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.

[3] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 33:1–33:15, Germany, 2016. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[4] Marco Carbone, Sonia Marin, and Carsten Schürmann. Forwarders as process compatibility, logically. *CoRR*, abs/2112.07636, 2021.

[5] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. In *CONCUR*, pages 412–426, 2015.

[6] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012.

[7] Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021.

[8] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, pages 22–138, 1998.

[9] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008.

[10] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *JACM*, 63(1):9, 2016. Also: POPL, 2008, pages 273–284.

[11] Julien Lange and Nobuko Yoshida. Verifying asynchronous interactions via communicating session automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2019.

[12] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[13] Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012.

[14] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2–3):384–418, 2014. Also: ICFP, pages 273–286, 2012.

From iterated parametricity to indexed semi-simplicial and semi-cubical sets: a formal construction

Hugo Herbelin and Ramkumar Ramachandra

[1] IRIF, CNRS, Université de Paris Cité, Inria, France
hugo.herbelin@inria.fr
[2] Université de Paris Cité, France
r@artagnon.com

Abstract

The talk will remind how iterated parametricity connects to augmented semi-simplicial sets (unary case) and semi-cubical sets (binary case) and discuss the alternative between the fibered and indexed representations of parametricity. A construction of iterated parametricity in indexed form has been fully formalised in Coq which the talk will discuss.

An *augmented semi-simplicial set* is a presheaf from the category of strictly increasing functions over finite (possibly empty) totally ordered sets, that is, relying on the characterisation of strictly increasing functions from the most atomic ones, a family of sets X_n with face maps $d^n_i : X_{n+1} \to X_n$ for all $i \leq n$ such that $d^n_i \circ d^{n+1}_{j+1} = d^n_j \circ d^{n+1}_i$ for $i \leq j$. Let us call this definition the *fibered* definition of augmented semi-simplicial sets and consider the alternative definition obtained by iteratively applying the equivalence between functions to a set and families of sets indexed over this set (a degenerate form of Grothendieck's construction), that is, type-theoretically, between $\Sigma B : \mathsf{hSet}.(B \to A)$ and $A \to \mathsf{hSet}$ for $A : \mathsf{hSet}$. We call *indexed* definition of augmented semi-simplicial sets the alternative definition given by a family:

$$\begin{array}{lcl} X_0 & : & \mathsf{hSet} \\ X_1 & : & X_0.\ \to \mathsf{hSet} \\ X_2 & : & \Pi a_0 : X_0.\ X_1 a_0 \to X_1 a_0 \to \mathsf{hSet} \\ X_3 & : & \Pi a_0 : X_0.\ \Pi a_1 b_1 c_1 : X_1 a_0.\ X_2 a_0 a_1 b_1 \to X_2 a_0 a_1 c_1 \to X_2 a_0 b_1 c_1 \to \mathsf{hSet} \\ \vdots & & \end{array}$$

where, up to isomorphism, each declaration takes the form $X_{n+1} : \mathsf{frame}^n(X_0, ..., X_n) \to \mathsf{hSet}$ for a well-chosen definition of frame^n.

A possible recursive definition of frame^n was given in [Her15] (see also [Voe12, CK21]) but an alternative definition inspired from iterated parametricity translation can be given too. Indeed, it is known that the iteration of Reynolds' binary parametricity translation yields an indexed definition of semi-cubical sets, as sketched in [HM20] and studied in a categorical setting by Moeneclaey [Moe21] (about the relation between cubical sets and iterated parametricity, see also e.g. [AK18, GJF+15, JS17, CH20]). Moeneclaey[1] also noticed that the indexed definition of augmented semi-simplicial sets actually corresponds to the iteration of the unary version of the parametricity translation. The current talk is about giving the construction from [HM20] in all details, supported by a full formalisation in Coq, covering both the unary (that is augmented semi-simplicial) and binary (that is semi-cubical) cases.

[1] private communication

The informal intuition behind the construction was given in [HM20] which will be reminded in the talk. Fixing a universe level l, we now shortly explain the formal construction. It goes by inductively defining n-truncated sets $X_l^{<n} : \mathsf{U}_{l+1}$:

$$\begin{array}{lcl} X_l^{<0} & \triangleq & \mathsf{unit} \\ X_l^{<n'+1} & \triangleq & \Sigma D : X_l^{<n'}.\, (\mathsf{frame}_l^n(D) \to \mathsf{hSet}_l) \end{array}$$

then take the coinductive closure of it: $X_l \triangleq X_l^{\geq 0}(\star)$ where $\star : \mathsf{unit}$ defines the unit type, and, coinductively, $X_l^{\geq n}(D) \triangleq \Sigma X : (\mathsf{frame}_l^n(D) \to \mathsf{hSet}_l).\, X_l^{\geq n+1}(D, X)$. Itself, $\mathsf{frame}_l^n(D) \triangleq \mathsf{frame}_l^{n,n}(D) : \mathsf{hSet}_l$ is defined inductively by layers:

$$\begin{array}{llcl} \mathsf{frame}_l^{n,0} & \star & \triangleq & \mathsf{unit} \\ \mathsf{frame}_l^{n,p'+1} & D & \triangleq & \Sigma d : \mathsf{frame}_l^{n,p'}(D).\, \mathsf{layer}_l^{n,p'}(d) \end{array}$$

where, for N the arity of the translation (here $N = 1$ or $N = 2$), a layer is made of N filled frames:

$$\mathsf{layer}_l^{n,p}(d) \triangleq \Pi\varepsilon : [1, N].\, \mathsf{filler}_l^{n-1,p}(\mathsf{restr}^{n,p}_{\mathsf{frame},l,\varepsilon,p}(d))$$

where, for $D : X_l^{<n}$ and $E : \mathsf{frame}_l^n(D) \to \mathsf{hSet}_l$, we have that $\mathsf{filler}_l^{n,p} : \mathsf{frame}_l^{n,p}(D) \to \mathsf{hSet}_l$ is itself defined by reverse induction from p to n by:

$$\begin{array}{llcl} \mathsf{filler}_l^{n,p,[p=n]} & (d) & \triangleq & E(d) \\ \mathsf{filler}_l^{n,p,[p<n]} & (d) & \triangleq & \Sigma l : \mathsf{layer}_l^{n,p}(d).\, \mathsf{filler}_l^{n,p+1}(d, l) \end{array}$$

In the definition of layers, for $D : X_l^{<n}$, $E : \mathsf{frame}_l^n(D) \to \mathsf{hSet}_l$ and $d : \mathsf{frame}_l^{n,p}(D)$, a family of restriction operators following the inductive structure of frames and playing the role of q-ε-face for the indexed construction is used:

$$\begin{array}{l} \mathsf{restr}^{n,p,[p\leq q<n]}_{\mathsf{frame},l,\varepsilon,q} : \mathsf{frame}_l^{n,p}(D) \to \mathsf{frame}_l^{n-1,p}(D.1) \\ \mathsf{restr}^{n,p,[p<q<n]}_{\mathsf{layer},l,\varepsilon,q} : \mathsf{layer}_l^{n,p}(d) \to \mathsf{layer}_l^{n-1,p}(\mathsf{restr}^{n,p}_{\mathsf{frame},l,\varepsilon,q}(d)) \\ \mathsf{restr}^{n,p,[p\leq q<n]}_{\mathsf{filler},l,\varepsilon,q} : \mathsf{filler}_l^{n,p}(d) \to \mathsf{filler}_l^{n-1,p}(\mathsf{restr}^{n,p}_{\mathsf{frame},l,\varepsilon,q}(d)) \end{array}$$

The key case is

$$\mathsf{restr}^{n,p,[p=q]}_{\mathsf{filler},l,\varepsilon,q}(b, \_) \triangleq b\,\varepsilon$$

but it also requires for $\mathsf{restr}_{\mathsf{layer},l}$ a coherence condition showing

$$\mathsf{restr}^{n-1,p}_{\mathsf{frame},l,\varepsilon,q}(\mathsf{restr}^{n,p}_{\mathsf{frame},l,\omega,r}(d)) = \mathsf{restr}^{n-1,p}_{\mathsf{frame},l,\omega,r}(\mathsf{restr}^{n,p}_{\mathsf{frame},l,\varepsilon,q+1}(d))$$

for $r \leq q$. This is also proved by following the inductive structure of frames and it eventually holds because we reason in hSet.

Taking into account coherences, the construction at level n requires to know what has been built at level $n-1$, $n-2$ and $n-3$. Working by full well-founded induction turned out to be difficult, so, in the formalisation[2], we restrict ourselves to maintain only the needed last 3 levels at each stage. Note that the formalisation takes great benefit of Coq's strict Prop to equate all syntactically distinct proofs of a given inequality occurring in the development. It additionally uses a representation of inequality proofs "à la Yoneda" (i.e. $n \leq_{\mathcal{Y}} p \triangleq \Pi q.\, q \leq n \to q \leq p$) to take benefit of even more definitional equalities in the proof.

[2] https://github.com/artagnon/bonak

References

[AK18] Thorsten Altenkirch and Ambrus Kaposi. Towards a cubical type theory without an interval. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 3:1–3:27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[CH20] Evan Cavallo and Robert Harper. Internal Parametricity for Cubical Type Theory. In Maribel Fernández and Anca Muscholl, editors, *28th EACSL Annual Conference on Computer Science Logic (CSL 2020)*, volume 152 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[CK21] Joshua Chen and Nicolai Kraus. Semisimplicial types in internal categories with families, June 2021. Talk at TYPES 2021 - http://www.cs.nott.ac.uk/ psznk/docs/internalsemisimp.pdf.

[GJF+15] Neil Ghani, Patricia Johann, Fredrik Nordvall Forsberg, Federico Orsanigo, and Tim Revell. Bifibrational functorial semantics of parametric polymorphism. *Electronic Notes in Theoretical Computer Science*, 319:165–181, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[Her15] Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, 25:1116–1131, 6 2015.

[HM20] Hugo Herbelin and Hugo Moeneclaey. Investigations into syntactic iterated parametricity and cubical type theory, 2020. Workshop on Homotopy Type Theory/ Univalent Foundations.

[JS17] Patricia Johann and Kristina Sojakova. Cubical categories for higher-dimensional parametricity. *CoRR*, abs/1701.06244, 2017.

[Moe21] Hugo Moeneclaey. Parametricity and semi-cubical types. In *LICS*, pages 1–11. IEEE, 2021.

[Voe12] Vladimir Voevodsky. Semi-simplicial types, November 2012. Online at http://uf-ias-2012.wikispaces.com/Semi-simplicial+types.

Higher-Order Universe Operators in Martin-Löf Type Theory with one Mahlo Universe

Yuta Takahashi

Ochanomizu University, Tokyo, Japan
takahashi.yuta@is.ocha.ac.jp

Background. Martin-Löf type theory has the so-called *universe types*. To provide the type-theoretic formulation of the Mahlo property, Setzer extended Martin-Löf type theory by means of a large universe type: he introduced Martin-Löf type theory **MLM** with one *Mahlo universe*, and determined its proof-theoretic ordinal [8, 9]. Mahlo universes have a reflection property similar to the ones of weakly Mahlo cardinals and recursively Mahlo ordinals.

On the other hand, Rathjen, Griffor and Palmgren [7] extended Martin-Löf type theory in another way: they introduced a system **MLQ** of Martin-Löf type theory. The system **MLQ** has the two types M and Q: roughly speaking, Q is an inductively defined set of codes for operators which provides universes closed under the universe operators constructed previously, and M is a universe closed under operators in Q. Using Aczel's interpretation [1, 2, 3] of constructive set theory **CZF** in Martin-Löf type theory, Rathjen, Griffor and Palmgren showed that **CZF** with an axiom asserting the existence of inaccessible sets of all transfinite orders is interpretable in **MLQ**. Moreover, the inductive construction of M and Q in **MLQ** was generalised by Palmgren [4]. He introduced a family $\mathbf{ML}^n$ of systems of *higher-order universe operators*, and showed that **MLQ** is an instance of these systems.

In sum, two powerful extensions of Martin-Löf type theory were introduced so far to study the type-theoretic counterparts of large sets: the extension by means of a reflection property similar to Mahloness (e.g. **MLM**), and the extension by means of higher-order universe operators (e.g. **MLQ** and $\mathbf{ML}^n$). The comparison between these two extensions in terms of their proof-theoretic strength was already attempted in the literature (see, for example, [4, 6, 9]). However, a more direct examination of the relationship between them is desirable as well.

Aim and Approach. We investigate the relationship between Mahlo universes and higher-order universe operators. Specifically, we show that higher-order universe operators in **MLQ** can be simulated in **MLM**, and extend this simulation to more general cases in $\mathbf{ML}^n$.

Below we use the logical framework adopted in the proof assistant Agda. We also adopt the families-of-sets formulation of **MLM** in [5], since this enables one to see the connection between Mahlo universes and higher-order universe operators more easily. Informally, a Mahlo universe type $\mathrm{V} : \mathrm{Set}$ "reflects" any operator on families of sets in V: for any $f : (\sum_{(a:\mathrm{V})} \mathrm{T}\, a \to \mathrm{V}) \to (\sum_{(a:\mathrm{V})} \mathrm{T}\, a \to \mathrm{V})$, where $\mathrm{T} : \mathrm{V} \to \mathrm{Set}$ is the decoding function for V, there is a subuniverse U_f of V with the decoding function $\widehat{\mathrm{T}}_f : \mathrm{U}_f \to \mathrm{V}$ such that U_f is closed under f. That is, we have

$$\frac{\Gamma \vdash f : (\sum_{(a:\mathrm{V})} \mathrm{T}\, a \to \mathrm{V}) \to (\sum_{(a:\mathrm{V})} \mathrm{T}\, a \to \mathrm{V})}{\Gamma \vdash \widehat{\mathrm{U}}_f : \mathrm{V}}\ \widehat{\mathrm{U}}\text{-}I \qquad\qquad \mathrm{T}\, \widehat{\mathrm{U}}_f = \mathrm{U}_f$$

The closedness property of U_f can be explained as follows. Define $\mathrm{T}_f : \mathrm{U}_f \to \mathrm{Set}$ as $\mathrm{T}_f a := \mathrm{T}(\widehat{\mathrm{T}}_f a)$. Then, the closedness under f is expressed by the operator $\mathsf{Res}_f : (\sum_{(a:\mathrm{U}_f)} \mathrm{T}_f a \to \mathrm{U}_f) \to (\sum_{(a:\mathrm{U}_f)} \mathrm{T}_f a \to \mathrm{U}_f)$ in **MLM**. This operator has the computation rule saying that

$\iota_f(\mathsf{Res}_f(a,b)) = f\ (\iota_f(a,b))$ holds for any $(a,b) : \sum_{(x:\mathrm{U}_f)} \mathrm{T}_f x \to \mathrm{U}_f$, where the injection $\iota_f : (\sum_{(x:\mathrm{U}_f)} \mathrm{T}_f x \to \mathrm{U}_f) \to (\sum_{(x:\mathrm{V})} \mathrm{T}\, x \to \mathrm{V})$ is defined as $\iota_f(a,b) := (\widehat{\mathrm{T}}_f a, \lambda x.\widehat{\mathrm{T}}_f(b\,x))$. The operator Res_f is intended as the restriction of f to the subuniverse U_f of V, and the injection ι_f shows that Res_f is indeed the restriction of f to U_f.

This reflection property enables Mahlo universes to define various universe operators. Before higher-order universe operators, we consider three examples: a universe above a family of sets in V, a usual universe operator and a super universe. A universe U above arbitrary $a : \mathrm{V}$ and $b : \mathrm{T}a \to \mathrm{V}$ is obtained by defining $f_0 : (\sum_{(x:\mathrm{V})} \mathrm{T}x \to \mathrm{V}) \to (\sum_{(x:\mathrm{V})} \mathrm{T}x \to \mathrm{V})$ as $f_0 := \lambda c.(a,b)$, where c does not occur in a nor b freely. Let $\widehat{\mathrm{N}}_{0,f_0}$ (resp. $\widehat{\mathrm{N}}_0$) be the code in U_{f_0} (resp. V) for the empty type, and C_0 be the eliminator for the empty type.

$$\iota_{f_0}(\mathsf{Res}_{f_0}(\widehat{\mathrm{N}}_{0,f_0}, \lambda x.\mathsf{C}_0 x)) = f_0(\iota_{f_0}(\widehat{\mathrm{N}}_{0,f_0}, \lambda x.\mathsf{C}_0 x)) = (\lambda c.(a,b))(\widehat{\mathrm{N}}_0, \lambda x.\widehat{\mathrm{T}}_{f_0}(\mathsf{C}_0 x)) = (a,b).$$

Therefore, the left projection $\mathsf{p}_1(\mathsf{Res}_{f_0}(\widehat{\mathrm{N}}_{0,f_0}, \lambda x.\mathsf{C}_0 x))$ of type U_{f_0} is a code in U_{f_0} for a, and the right projection $\mathsf{p}_2(\mathsf{Res}_{f_0}(\widehat{\mathrm{N}}_{0,f_0}, \lambda x.\mathsf{C}_0 x))$ is a code for b.

We define a super universe by reflecting a usual universe operator. To see this, note that an operator f which is reflected by V may have some parameters. For instance, when we have

$$y : \textstyle\sum_{(x:\mathrm{V})} \mathrm{T}\, x \to \mathrm{V} \vdash \lambda c.y : (\sum_{(x:\mathrm{V})} \mathrm{T}\, x \to \mathrm{V}) \to (\sum_{(x:\mathrm{V})} \mathrm{T}\, x \to \mathrm{V}),$$

we first obtain U_f with $f := \lambda c.y$ by the $\widehat{\mathrm{U}}$-I rule above. Then, we have the universe operator $\lambda y.(\widehat{\mathrm{U}}_f, \widehat{\mathrm{T}}_f) : (\sum_{(x:\mathrm{V})} \mathrm{T}\, x \to \mathrm{V}) \to (\sum_{(x:\mathrm{V})} \mathrm{T}\, x \to \mathrm{V})$ mapping any family of sets in V to a universe above this family. By the $\widehat{\mathrm{U}}$-I rule again, we obtain $\widehat{\mathrm{U}}_g : \mathrm{V}$ with $g := \lambda y.(\widehat{\mathrm{U}}_f, \widehat{\mathrm{T}}_f)$. Since g is a universe operator and U_g is closed under g, the universe U_g is a super universe.

One can treat higher-order universe operators as universe operators of type $\mathcal{O}_n$ with $n > 1$ in the sense of [4, Definition 5.1], where $\mathcal{O}_0 := \mathrm{Set}$, $\mathcal{F}_n := \sum_{(A:\mathrm{Set})} A \to \mathcal{O}_n$ and $\mathcal{O}_{n+1} := \mathcal{F}_n \to \mathcal{F}_n$. Note that $\mathcal{O}_1$ is the type of "first-order" operators, which are mappings from families of sets to themselves. Since our aim is to construct higher-order universe operators by means of the Mahlo universe V, we consider higher-order universe operators as universe operators of type O_n with $n > 1$, where $\mathrm{O}_0 := \mathrm{V}$, $\mathrm{F}_n := \sum_{(a:\mathrm{V})} \mathrm{T}\, a \to \mathrm{O}_n$ and $\mathrm{O}_{n+1} := \mathrm{F}_n \to \mathrm{F}_n$. We call an operator in O_n an *operator of order* n. For instance, the operator g above is of type O_1.

Higher-order universe operators in **MLQ** (see [7, § 3.2]) are, in our setting, operators whose inputs are a family (a_1, b_1) of type F_1 and a family (a_0, b_0) of type F_0. Then, these higher-order operators return a universe $\mathrm{U}(a_1, b_1, a_0, b_0)$ which is closed under the operator $b_1 x$ for each $x : \mathrm{T}\, a_1$, and includes (the codes of) (a_0, b_0). It is straightforward to transform these operators into universe operators of order 2 in the sense above, as explained in [4, Example 5.4].

Our idea for constructing higher-order universe operators is to use parameters in reflecting operators similarly to the case of the universe operator g above. As we utilised a family of sets in V as a parameter y in the case of g, the simulation of higher-order universe operators of **MLQ** requires a family of operators of order 1 as a parameter. Consider the following variables: $x : \mathrm{F}_1$, $e : \mathrm{T}(\mathsf{p}_1 x)$, $y : \mathrm{F}_0$ and $z : \mathrm{N}_2$ with the boolean type N_2. We define the operators h_0, h_1, h of order 1 as $h_0 := \lambda c.y$, $h_1 := \mathsf{p}_2 x\ e$ and $h := \mathsf{C}_2\ z\ h_0\ h_1$, where C_2 is the eliminator for N_2. Then, we have a universe $\widehat{\mathrm{U}}_h$ by reflection, and the family below can be defined:

$$(\widetilde{\mathrm{U}}, \widetilde{\mathrm{T}}) := \left(\widehat{\Sigma}\ (\mathsf{p}_1 x)\ (\lambda e.(\widehat{\Sigma}\ \widehat{\mathrm{N}}_2(\lambda z.\widehat{\mathrm{U}}_h))),\ \ \lambda v.\widehat{\mathrm{T}}_{h'}(\mathsf{p}_2(\mathsf{p}_2 v))\right) : \mathrm{F}_0 \text{ with } h' = h[\mathsf{p}_1 v/e][\mathsf{p}_1(\mathsf{p}_2 v)/z].$$

This family and Res_h simulate a universe which is closed under $\mathsf{p}_2 x\ e$ for each $e : \mathrm{T}\,(\mathsf{p}_1 x)$ and includes the family y. Finally, we obtain the higher-order operator $\lambda x.(\widehat{\mathrm{N}}_1, \lambda w.\lambda y.(\widetilde{\mathrm{U}}, \widetilde{\mathrm{T}})) : \mathrm{O}_2$, which takes arbitrary $x : \mathrm{F}_1$ and returns a universe closed under the operators in x. Operators of greater order in $\mathbf{ML}^n$ can be simulated as well by using parameters of greater order.

References

[1] Peter Aczel. The type theoretic interpretation of constructive set theory. In Angus Macintyre, Leszek Pacholski, and Jeff Paris, editors, *Logic Colloquium '77*, volume 96 of *Studies in Logic and the Foundations of Mathematics*, pages 55–66. Elsevier, 1978.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory: Choice principles. In A. S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, pages 1–40. North-Holland, 1982.

[3] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definitions. In R. B. Marcus, G. J. Dorn, and G. J. W. Dorn, editors, *Logic, Methodology, and Philosophy of Science VII*, pages 17–49. North-Holland, 1986.

[4] 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.

[5] Michael Rathjen. Realizing Mahlo set theory in type theory. *Arch. Math. Log.*, 42(1):89–101, 2003.

[6] Michael Rathjen. The constructive Hilbert program and the limits of Martin-Löf type theory. *Synth.*, 147(1):81–120, 2005.

[7] Michael Rathjen, Edward R. Griffor, and Erik Palmgren. Inaccessibility in constructive set theory and type theory. *Ann. Pure Appl. Log.*, 94(1-3):181–200, 1998.

[8] Anton Setzer. Extending Martin-Löf type theory by one Mahlo-universe. *Arch. Math. Log.*, 39(3):155–181, 2000.

[9] Anton Setzer. Universes in type theory part I – Inaccessibles and Mahlo. In A. Andretta, K. Kearnes, and D. Zambella, editors, *Logic Colloquium '04*, pages 123–156. Association of Symbolic Logic, Lecture Notes in Logic 29, Cambridge University Press, 2008.

Homotopy setoids and quotient completion

Cipriano Junior Cioffo

Università degli Studi di Milano

Introduction. A *setoid* is a notion of *set* in constructive mathematics, originally introduced by Bishop in [2]. In Martin-Löf intuitionistic type theory [13] a setoid can be described as a pair (X, R) where X is a closed type and R is a dependent type of the form

$$x_1, x_2 : \mathrm{X} \vdash \mathrm{R}(x_1, x_2)$$

satisfying suitable reflexivity, symmetry and transitivity conditions. The importance of setoids in type theory has been deeply investigated by Hofmann in [8] who introduced the *setoid model* in order to obtain *extensional constructs* in *intensional* type theory. Mainly, setoids are used to provide the *quotient* construction in place of the *quotient types* which lead to the undecidability of the type check or the introduction of non-canonical elements.

In category theory, setoids provide an instance of the categorical construction called the *exact completion* which is briefly described as follows. Given a weakly left exact category (wlex) $\mathscr{C}$, the exact completion $\mathscr{C}_{ex}$ of $\mathscr{C}$ is an exact category in the sense of Barr [1]. The objects of $\mathscr{C}_{ex}$ are given by the *pseudo equivalence relations* i.e. pairs of arrows of $\mathscr{C}$

$$r_1, r_2 : R \to X$$

satisfying suitable reflexivity, symmetry and transitivity conditions. An arrow between two pseudo relations $r_1, r_2 : R \to X$ and $s_1, s_2 : S \to Y$ is the equivalence class of a pair of arrows $(f, \tilde{f})$, where $f : X \to Y$ and $\tilde{f} : R \to S$, such that $f \circ r_i = s_i \circ \tilde{f}$, for $i = 1, 2$. The equivalence relation is given by the notion of *half-homotopy*, see [4]. Closed types and functions up to *functional extensionality* form a category **ML** which has strict finite products and weak pullbacks when the type theory considered is intensional. Hence, **ML** is a wlex category. The category **Std** of setoids and functions compatible with the relations has been widely studied, and it turns out to be equivalent to the exact completion of the category **ML**.

In order to study the properties of the exact completion $\mathscr{C}_{ex}$, one can verify if the category $\mathscr{C}$ shares a weaker version of these properties. For instance, in [3, Theorem 3.3] and [5, Theorem 3.6] the authors characterize the categories whose exact completion leads to a *local cartesian closed* category (lcc). In [7, Proposition 2.1], the authors characterize the categories whose exact completion is an *extensive* category. These results apply to the category of setoids which was already known to a be an lcc pretopos. Hence, we can summarize the above discussion in the following well-known results.

Fact. *The category* ***Std*** *is an lcc pretopos and* $\boldsymbol{ML}_{ex} \cong$ ***Std***.

Different parts of the proof of the first part of this fact can be found in [14] and [6].

Homotopy setoids. We have considered a homotopical version of setoids in view of ideas from the homotopy type theory [15]. A *homotopy setoids* is a setoid (X, R) such that the base type X is an *h-set* and the equivalence relation R is an *h-proposition*. By definition, h-props and h-sets are types such that the following types are inhabited

$$\mathsf{is\text{-}prop}(R) := \prod_{x,y:R} \mathsf{Id}_R(x, y) \qquad \mathsf{is\text{-}set}(X) := \prod_{x,y:X} \mathsf{is\text{-}prop}(\mathsf{Id}_X(x, y)).$$

Intuitively, h-propositions are types that are empty or contractible and h-sets are types that are *discrete*. The category $\mathbf{Std}_0$ is the full subcategory of $\mathbf{Std}$ of h-setoids and functions preserving relations. Our main objective is to prove that $\mathbf{Std}_0$ shares properties similar to $\mathbf{Std}$.

Problem. *The category $\boldsymbol{Std}_0$ is not exact and hence it is not the exact completion of a suitable category.*

A possible solution to this problem is to study homotopy setoids in the more general context of *elementary doctrines*. We recall that doctrines were introduced by Lawvere [9] and provide a categorical tool to work with syntactic logical theories. A *primary* doctrine is a functor $\mathsf{P} : \mathscr{C}^{op} \to \mathsf{Pos}$ form a category $\mathscr{C}$ with finite products to the category of *partially ordered sets (posets)* and monotone functions. For instance, we can consider the functor $F^{ML} : \mathbf{ML}^{op} \to \mathsf{Pos}$ which associate to each closed type X, the poset of the types depending on X up to *logical equivalence*, i.e. two types A(x) and B(x) are equivalent when the type

$$x : X \vdash (A(x) \Rightarrow B(x)) \times (B(x) \Rightarrow A(x))$$

is inhabited. The action of F^{ML} on arrows is given by substitution of terms. If we denote with $\mathbf{ML}_0$ the full subcategory of $\mathbf{ML}$ of h-sets, then we can similarly consider the functor $F^{ML_0} : \mathbf{ML}_0^{op} \to \mathsf{Pos}$ which sends an h-set X to the poset of the types depending on X which are h-propositions. Actually, the above functors are *elementary* doctrines. These structures were introduced by Maietti and Rosolini in [11] and they are suitable primary doctrines which can deal with the equality predicate. This is achieved requiring, for each object $X \in \mathscr{C}$, the existence of an element $\delta_X \in \mathsf{P}(X \times X)$ such that

1. $\top_X \leq \mathsf{P}_{\Delta_X} \delta_X$
2. $\mathsf{P}_{p_1} \alpha \wedge \delta_X \leq \mathsf{P}_{p_2} \alpha$, for every $\alpha \in \mathsf{P}(X)$
3. $\mathsf{P}_{\langle p_1, p_3 \rangle} \delta_X \wedge \mathsf{P}_{\langle p_2, p_4 \rangle} \delta_y \leq \delta_{X \times Y}$, for every pair of objects $X, Y \in \mathscr{C}$.

For the elementary doctrines F^{ML} and F^{ML_0} the role of the equality is played by the identity type Id_X.

If P is an elementary doctrine, it is possible to define P-eq. relations and the corresponding notion of well-behaved quotients. In [11, 10, 12], the authors provide a construction which associates to each elementary doctrine P an elementary doctrine $\overline{\mathsf{P}} : \overline{\mathscr{C}}^{op} \to \mathsf{Pos}$, called the *elementary quotient completion* of P, with well-behaved quotients in a suitable universal way. The base category $\overline{\mathscr{C}}$ has objects given by pairs (X, ρ) with ρ a P-eq. relation on X. The arrows of $\overline{\mathscr{C}}$ are given by those arrows of $\mathscr{C}$ which preserve the P-equivalence relations. By construction, $\overline{\mathscr{C}}$ has quotients of all $\overline{\mathsf{P}}$-equivalence relations.

Example. The following are examples of elementary quotient completion:

1. The exact completion of a category with finite products and weak pullbacks is an instance of the elementary quotient completion for the elementary doctrine of *weak subobjects*.
2. The category $\mathbf{Std}$ is equivalent to the base category $\overline{\mathbf{ML}}$ and the category $\mathbf{Std}_0$ is equivalent to the base category $\overline{\mathbf{ML}_0}$.

We have provided a version of the [3, Theorem 3.3] and [7, Proposition 2.1] in the context of elementary doctrines and elementary quotient completion which generalize the statements for categories with strict products and weak pullbacks. Hence, we could define *relative pretoposes* as the elementary doctrines $\mathsf{P} : \mathscr{C}^{op} \to \mathsf{Pos}$ with well-behaved quotients such that the category $\mathscr{C}$ is extensive. In this case, we say that the category $\mathscr{C}$ is a pretopos relative to P. We applied the results to F^{ML_0} and we obtained the following property for h-setoids.

Theorem. *The category $\boldsymbol{Std}_0$ is a lcc pretopos relative to $\overline{F^{ML_0}}$.*

References

[1] Michael Barr. Exact categories. In *Exact categories and categories of sheaves*, pages 1–120. Springer, 1971.

[2] Errett Bishop. Foundations of constructive analysis. 1967.

[3] Aurelio Carboni and Giuseppe Rosolini. Locally cartesian closed exact completions. *Journal of Pure and Applied Algebra*, 154(1-3):103–116, 2000.

[4] Aurelio Carboni and Enrico M Vitale. Regular and exact completions. *Journal of pure and applied algebra*, 125(1-3):79–116, 1998.

[5] Jacopo Emmenegger. On the local cartesian closure of exact completions. *Journal of Pure and Applied Algebra*, page 106414, 2020.

[6] Jacopo Emmenegger and Erik Palmgren. Exact completion and constructive theories of sets. *The Journal of Symbolic Logic*, 85(2):563–584, 2020.

[7] Marino Gran and EM Vitale. On the exact completion of the homotopy category. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 39(4):287–297, 1998.

[8] Martin Hofmann. Extensional concepts in intensional type theory. 1995.

[9] F William Lawvere. Adjointness in foundations. *Dialectica*, pages 281–296, 1969.

[10] Maria Emilia Maietti and Giuseppe Rosolini. Elementary quotient completion. *Theory and applications of categories*, 27(17):445–463, 2013.

[11] Maria Emilia Maietti and Giuseppe Rosolini. Quotient completion for the foundation of constructive mathematics. *Logica Universalis*, 7(3):371–402, 2013.

[12] Maria Emilia Maietti and Giuseppe Rosolini. Unifying exact completions. *Applied Categorical Structures*, 23(1):43–52, 2015.

[13] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[14] Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1-3):189–218, 2000.

[15] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

Implementing Martin-Löf's Meaning Explanations for Intuitionistic Type Theory in Agda

Peter Dybjer[1] and Anton Setzer[2]

[1] Chalmers University of Technology, Gothenburg, Sweden, Dept. of Computer Science and Engineering
peterd@chalmers.se
http://www.cse.chalmers.se/~peterd/
[2] Swansea University, Dept. of Computer Science
a.g.setzer@swansea.ac.uk
http://www.cs.swan.ac.uk/~csetzer/

At the Congress of LMPS in Hannover in 1979 Martin-Löf presented his seminal paper "Constructive Mathematics and Computer Programming" [1]. In this paper there is not only a new version of intuitionistic type theory but it also contained so called "meaning explanations" for this theory. The meaning explanations can be viewed both meta-mathematically, as a certain kind of model construction in the tradition of Kleene realizability, and philosophically, as a "pre-mathematical" discussion of the direct, intuitive semantics of type theory.

In this talk we will discuss both aspects, and present a formalization of meaning explanations in Agda (work in progress). More specifically, we will formalize the syntax and computation rules of a fragment of intuitionistic type theory, in order to make meaning explanations written in natural language more precise.

The computation relation relates an untyped term to its canonical form, where a canonical form is a term the outermost form of which is a constructor. The meaning of the judgements of type theory is then defined in terms of such computation to canonical form. Finally, we prove the correctness of some of the inference rules of intuitionistic type theory. In particular we show that the rules of extensional identity types are validated.

Meaning explanations represent an important contribution to the foundations of intuitionistic logic and constructive mathematics. They clarify the Brouwer-Heyting-Kolmogorov interpretations of the logical constants and also generalize them. They do not only explain the meaning of the logical constants but also provide a conception of what a constructive mathematical object is in general.

From a "pre-mathematical" point of view, the aim of the meaning explanations is to make the correctness of each rule of intuitionistic type theory immediately evident. In "Constructive Mathematics and Computer Programming" Martin-Löf famously finished with the following statement:

> For each of the rules of inference, the reader is asked to make the conclusion evident to himself on the presupposition that he knows the premises. This does not mean that further verbal explanations are of no help in bringing about an understanding of the rules, only that this is not the place for such detailed explanations. But there are also certain limits to what verbal explanations can do when it comes to justifying axioms and rules of inference. In the end, everybody must understand for himself.

Such further informal verbal explanations were provided in Martin-Löf's 1984 book "Intuitionistic Type Theory" [2]. In our formalization in Agda we prove formally the correctness of the inference rules in full detail. This provides rigorous justifications for them, and we find it

interesting to carry out those details. An essential aspect of the work is of course that we use a fragment of Agda far more complex than the theory we are justifying. In particular, we use features such as indexed inductive-recursive definitions and Agda's pattern matching, to make a transparent formalization.

Martin-Löf's meaning explanations provide the perhaps most convincing foundations of (predicative) constructive mathematics. To properly appreciate them one needs to understand both the technical formal meta-mathematical side, and the informal pre-mathematical side. We hope that our work will help in this endeavour.

References

[1] Per Martin-Löf. Constructive mathematics and computer programming. In L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski, editors, *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. North-Holland, 1982. doi:10.1016/S0049-237X(09)70189-2.

[2] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

Interpreting second-order arithmetic via update recursion

Valentin Blot

LMF, Inria, Université Paris-Saclay

Second-order arithmetic has two kinds of computational interpretations: via Spector's bar recursion [4] or via Girard-Reynolds polymorphic lambda-calculus [2, 3]. Bar recursion interprets the negative translation of the axiom of choice which, combined with an interpretation of the negative translation of the excluded middle, gives a computational interpretation of the negative translation of the axiom scheme of comprehension. It is then possible to instantiate universally quantified sets with arbitrary formulas (second-order elimination). On the other hand, polymorphic lambda-calculus interprets directly second-order elimination by means of polymorphic types. The present work aims at bridging the gap between these two interpretations by interpreting directly second-order elimination through update recursion, which is a variant of bar recursion due to Berger [1].

First, we show that a slight variant of Berger's update recursion [1] interprets the following principle:

$$\neg\neg\forall x\,(A\,(x) \vee \neg A\,(x))$$

which in turn implies the double negation of the axiom scheme of comprehension:

$$\neg\neg\exists X\forall x\,(X\,(x) \Leftrightarrow A\,(x))$$

The variant of Berger's update recursion that we use is:

$$\mathtt{ur} : ((\mathtt{nat} \to T + (T \to o)) \to o) \to (\mathtt{nat} \to T + \mathtt{unit}) \to o$$

and satisfies the following recursive equation:

$$\mathtt{ur}\,t\,u = t \left(\begin{array}{l} \lambda n.\mathtt{match}\,u\,n\,\mathtt{with} \\ \qquad \mathtt{inl}\,x \mapsto \mathtt{inl}\,x \\ \qquad \mathtt{inr}\,\_ \mapsto \mathtt{inr} \left(\lambda x.\mathtt{ur}\,t \left(\begin{array}{l} \lambda m.\mathtt{if}\,m = n \\ \quad \mathtt{then}\,\mathtt{inl}\,x \\ \quad \mathtt{else}\,u\,m \end{array} \right) \right) \\ \quad \mathtt{end} \end{array} \right)$$

If we see the type $T + \mathtt{unit}$ as an option type on T, $\mathtt{ur}$ provides t with an extension of u that performs a recursive call whenever $u\,n$ is not defined. We show that $\mathtt{ur}$ interprets the formula:

$$\neg\forall x\,(A\,(x) \vee \neg A\,(x)) \Rightarrow \neg\forall x\,(A\,(x) \vee \top)$$

and hence, feeding it with $\lambda\_.\mathtt{inr}\,\mathtt{tt}$ that interprets $\forall x\,(A\,(x) \vee \top)$, we obtain an interpretation for:

$$\neg\neg\forall x\,(A\,(x) \vee \neg A\,(x))$$

Second-order arithmetic can be obtained from first-order arithmetic by adding quantification over predicates. The logical power of second-order arithmetic resides in its second-order elimination rule:

$$\forall X\,B \Rightarrow B\,[A\,(x)\,/X\,(x)]$$

that instantiates a second-order variable X with an arbitrary formula $A(x)$. Using the `ur` operator above, we are able to define inductively on the structure of B a computational interpretation of the second-order elimination rule, and therefore of second-order arithmetic.

We define a bar recursive interpretation of second-order arithmetic presented as arithmetic with quantification on predicates rather than the equivalent axiom scheme of comprehension. This presentation of second-order arithmetic is the one that most closely reflects the typing rules of polymorphic λ-calculus, and as such we make a step towards a comparison of the two families of interpretations of second-order arithmetic: bar recursion and system F.

As a future work we would like to deepen the understanding of the connection between these two principles by comparing the computational behavior of programs extracted from a single proof via the two techniques.

Another aspect that we would like to study is wether it is possible to use control operators in the interpretation of the $\forall 2e$ rule. Indeed, there is a strong connection between the negative translation of proofs and the continuation-passing style (cps) translation of programs, the latter being the Curry-Howard equivalent of the former. Calculi with control features have been designed to interpret classical proofs directly. Most of these calculi contain a notion of duality that corresponds on the logical side to the duality between a formula and its negation, and on the computational side to a call-by-name or a call-by-value evaluation strategy. During its computation, update recursion has an asymmetric behavior that consists in building a realizer of $\neg B$ by reading a realizer of B and making a recursive call with a new knowledge extended with this new realizer. This behavior corresponds to the call-by-name interpretation of the excluded middle under a cps translation. We would therefore like to have a version of update recursion that uses control operators and can be translated either to the current version through a call-by-name cps translation, or to a dual version through a call-by-value cps translation. Moreover, control operators can capture context and restore it at a later point. We would like to explore the possibility of using this property to define more intuitive versions of our interpretation of second-order elimination that could act on all instances of $X(t)$ in a formula through context capture.

While termination of `ur` can be shown using Zorn's lemma, termination of the polymorphic lambda-calculus relies on impredicative reducibility candidates. A comparison would therefore provide a new approach to understanding impredicativity.

References

[1] Ulrich Berger. A computational interpretation of open induction. In *19th IEEE Symposium on Logic in Computer Science*, page 326. IEEE Computer Society, 2004.

[2] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–69. North-Holland, 1971.

[3] John Reynolds. Towards a theory of type structure. In *Programming Symposium, Paris, April 9-11, 1974*, Lecture Notes in Computer Science, pages 408–423. Springer, 1974.

[4] Clifford Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In *Recursive Function Theory: Proceedings of Symposia in Pure Mathematics*, volume 5, pages 1–27. American Mathematical Society, 1962.

Equivalence Between Typed and Untyped Conversion Algorithms

Meven Lennon-Bertrand

Inria – LS2N, Université de Nantes, Nantes, France

Abstract

This talk shall present a new take into the innocent-looking but difficult issue of relating the typed, judgmental and the untyped, rewriting-based approaches to conversion for dependent type theories, by considering algorithmic presentations of those.

1 A Tale of Two Conversions

An important component of dependent type systems is how they incorporate computation. This is usually done by means of a conversion rule, which allows replacing a type by one related to it by the conversion relation $\cong$. There are, however, two quite different approaches to defining that relation. In the first, that we will hereafter call *typed* conversion, it is presented by means of a judgement $\Gamma \vdash t \cong t' : A$, described by inference rules in much the same way as typing. The type is used in the rules, as showcased in the following examples:

$$\textsc{Fun}\ \frac{\Gamma, x : A \vdash f\ x \cong g\ x : B}{\Gamma \vdash f \cong g : \Pi x : A.B} \qquad \beta\ \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x : A.t)\ u : B[x := u]}$$

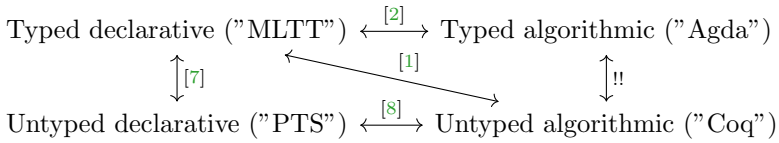
The second, that we call *untyped* conversion, is built on top of an untyped reduction relation: conversion is taken as the symmetric, reflexive, transitive closure of reduction.

The two definitions have been used concurrently for a long time. Typed conversion is a prominent characteristic of Martin-Löf Type Theory [5], and is the main source of inspiration for the implementation of Agda. Untyped conversion is the one favoured in Pure Type Systems (PTS) [3], and Coq follows this approach by implementing an untyped conversion routine.

While the two approaches are conceptually close, proving the equivalence between them is difficult. The best attempts to date are that of Siles and Herbelin [7], and of Abel and Coquand [1]. Siles and Herbelin's strategy is purely syntactic, and thus requires a weak meta-theory, but does not cover extensionality rules such as Fun above. Abel and Coquand handle these extensionality rules, but they need a meta-theory powerful enough to prove normalization of the object system. Thus, their approach applies only to systems that are not logically weak – they do not handle an impredicative sort of propositions, as present in Coq.

2 Enter the Algorithmic Presentations

These presentations of Section 1 – which we will call *declarative* – are quite remote from actual implementations. Thus, other presentations – that we call *algorithmic* – have been introduced, which aim to be much closer to the actual implementations of conversion checking than their declarative counterparts. In particular, they remove the transitivity rule, which is very problematic when trying to build an implementation. Such presentations have been given in [2] for typed conversion, and in MetaCoq [8] for untyped conversion, and in both cases proven equivalent to their declarative counterparts. The general picture thus looks like this:



Now, what about the equivalence in the right column? Can we prove it? With what tools, and what logical power? This is of particular interest because the equivalence of [8] does not currently handle extensionality rules. Indeed, to the best of our knowledge, giving an untyped, declarative presentation of Fun in a setting with other types than functions is an open problem. Proving the equivalence might give a way out, by moving the study to typed conversion where the picture is clearer – see [2] and its extensions [4, 6] –, and seeing the untyped algorithm as a mere optimization of the typed one. This way, we could relate a conversion-checking algorithm such as that of Coq to a typed specification, extensionality rules are less difficult to handle.

3 Proofs!

The equivalence. The two algorithmic conversions are built around the same idea, namely the interleaving of (weak-head) reduction and structural rules. Terms – and when applicable their type – are reduced to weak-head normal forms, and then structural rules may be applied based on the structure of those. The difference between the two systems resides in the way these structural rules work: in the typed variant, they are primarily type-directed – akin to Fun –, while the untyped variant only uses information available from the term. Showing the equivalence thus amounts to proving that both approaches give the same result, and so in particular that the type information is actually superfluous.

In the case of functions, what we need is that the two following rules, together with a symmetric version of λ-R and generic rules for neutral terms, can emulate Fun.

$$\lambda\text{-R}\quad \frac{\Gamma, x : A \vdash t \cong n\ x \qquad n \text{ is not a } \lambda}{\Gamma \vdash \lambda x : A.t \cong n} \qquad \lambda\text{-Cong}\quad \frac{\Gamma, x : A \vdash t \cong t'}{\Gamma \vdash \lambda x : A.t \cong \lambda x : A'.t'}$$

In both directions, the main difficulty is to show that the algorithm properly maintains well-typedness invariants. This in turn relies on standard meta-theoretical properties (reflexivity of conversion, stability by substitution, subject reduction...). In the direction from typed to untyped conversion, the key ingredient is then to show injectivity of η-expansion, that is that if $\Gamma, x : A \vdash f\ x \cong g\ x$ and both f and g have a function type in Γ, then $\Gamma \vdash f \cong g$. This is done by analysing the possible weak-head normal forms for both f and g. The other direction amounts to reconstructing the typing information, using the well-typedness invariants, since Fun directly subsumes all four undirected rules. A formalization is ongoing to check the details of this proof sketch.

What power do we need? All the required meta-theoretic properties are easy consequences of [2]. However, due to the very structured form of algorithmic conversion, most of them imply normalization, so they must be assumed if we want to keep using a low logical power in our proof of equivalence. A middle ground is to only assume one single property and derive all its consequences, as is done by *e.g.* MetaCoq, where a single normalization axiom is enough to fuel the whole development. Stability by substitution is strong enough to imply all other hypothesis, but it is currently unclear whether normalization would also suffice. Again, the ongoing formalization should provide a more solid setting to investigate this subtle point.

References

[1] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. *Fundamenta Informaticae*, 77(4):345–395, 2007. TLCA'05 special issue.

[2] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[3] Henk Barendregt. An introduction to generalized type systems. *Journal of Functional Programming*, 1:125–154, April 1991.

[4] 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, January 2019.

[5] Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory.* Number 1 in Studies in Proof Theory. Napoli: Bibliopolis, 1984.

[6] Loïc Pujet and Nicolas Tabareau. Observational equality: Now for good. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[7] Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *J. Funct. Program.*, 22(2):153–180, 2012.

[8] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

Linear lambda-calculus is linear*

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

[1] UNQ & ICC (CONICET–UBA), Argentina
[2] Inria & ENS Paris-Saclay, France

The name of linear logic [10] suggests that this logic has some relation with the algebraic notion of linearity. A common account of this relation is that a proof of a linear implication between two propositions A and B should not be any function mapping proofs of A to proofs of B, but a linear one. This idea has been fruitfully exploited to build models of linear logic (for example [3, 9, 11]), but it is seems difficult to even formulate it within the proof language itself. Indeed, expressing the properties $f(u+v) = f(u) + f(v)$ and $f(a.u) = a.f(u)$ requires an addition and a multiplication by a scalar, that are usually not present in proof languages.

The situation has changed with quantum programming languages [1,2,4,7,8,12,14] and with the algebraic λ-calculus [13], that mix some usual constructions of programming languages with algebraic operations. Several extensions of the lambda-calculus, or of a language of proof-terms, with addition and multiplication by a scalar have been proposed [2, 5, 13].

In this paper [6], we investigate an extension of linear logic with addition and multiplication by a scalar, the $\mathcal{L}^{\mathcal{S}}$-logic, and we prove a linearity theorem: if f is a proof of an implication between two propositions of some specific form, then $f(u+v) = f(u)+f(v)$ and $f(a.u) = a.f(u)$.

This work is part of a wider research program that aims at determining in which way propositional logic must be extended or restricted, so that its proof language becomes a quantum programming language. There are two main issues in the design of a quantum programming language: the first is to take into account the linearity of the unitary operators and, for instance, avoid cloning, and the second is to express the information-erasure, non-reversibility, and non-determinism of the measurement. In [5], we addressed the question of the measurement. In this paper [6], we address that of linearity.

Interstitial rules. To extend linear logic with addition and multiplication by a scalar, we proceed, like in [5, long version], in two steps: we first add interstitial rules and then scalars.

An interstitial rule is a deduction rule whose premises are identical to its conclusion. In the $\mathcal{L}^{\mathcal{S}}$-logic, we consider two such rules

$$\frac{\Gamma \vdash A \quad \Gamma \vdash A}{\Gamma \vdash A}\ \text{sum} \qquad\qquad \frac{\Gamma \vdash A}{\Gamma \vdash A}\ \text{prod}$$

These rules introduce constructors $\boldsymbol{+}$ and $\bullet$ in the proof language, that are used to express the addition and the multiplication by a scalar. The fact that these deduction rules are trivial expresses the fact that these operations are internal.

Adding these rules permits to build proofs that cannot be reduced, because the introduction rule of some connective and its elimination rule are separated by an interstitial rule. Reducing such a proof, sometimes called a commuting cut, requires reduction rules to commute the rule sum either with the elimination rule below or with the introduction rules above.

Scalars. We then consider a field $\mathcal{S}$ of scalars and replace the introduction rule of the connective $\top$ with a family of rules $\top$-i(a), one for each scalar, and the rule prod with a family of rules prod(a), also one for each scalar

$$\frac{}{\Gamma \vdash \top}\ \top\text{-i}(a) \qquad\qquad \frac{\Gamma \vdash A}{\Gamma \vdash A}\ \text{prod}(a)$$

*This is an extended abstract. The full paper can be found at [6]

The connective $\odot$. Besides interstitial rules and scalars, we have introduced, in [5, long version], a new connective $\odot$ (read "sup" for "superposition"), that has an introduction rule $\odot$-i similar to that of the conjunction, two elimination rules $\odot$-e1 and $\odot$-e2 similar to those of the conjunction, but also a third elimination rule $\odot$-e similar to that of the disjunction.

The elimination rules $\odot$-e1 and $\odot$-e2 are used to express the information-preserving, reversible, and deterministic operations, such as the unitary transformations of quantum computing. The elimination rule $\odot$-e is used to express the information-erasing, non-reversible, and non-deterministic operations, such as quantum measurement.

Starting from propositional logic with the interstitial rules sum and prod, we can thus either add scalars, or the connective $\odot$, or both. This yields the four logics: PL is propositional logic with the interstitial rules sum and prod, $\text{PL}^{\mathcal{S}}$ is propositional logic with the interstitial rules and scalars, $\odot$ is propositional logic with the interstitial rules and the connective $\odot$, and $\odot^{\mathcal{S}}$ is propositional logic with the interstitial rules, the connective $\odot$, and scalars.

Linearity. The proof language of the $\odot^{\mathcal{S}}$-logic is a quantum programming language, as quantum algorithms can be expressed in it. However, this language addresses the question of quantum measurement, but not the that of linearity, and non-linear functions, such as cloning operators, can also be expressed in it. This leads to introduce, in our paper [6], a linear variant of the $\odot^{\mathcal{S}}$-logic, and prove a linearity theorem for it. More generally, we can introduce a linear variant for each of the previously mentioned four logics: $\mathcal{L}$ is linear logic with the interstitial rules sum and prod, $\mathcal{L}^{\mathcal{S}}$ is linear logic with the interstitial rules and scalars, $\mathcal{L}\odot$ is linear logic with the interstitial rules and the connective $\odot$, and $\mathcal{L}\odot^{\mathcal{S}}$ is linear logic with the interstitial rules, the connective $\odot$, and scalars.

Our goal is to prove a linearity theorem for the proof language of the $\mathcal{L}\odot^{\mathcal{S}}$-logic. But such a theorem does not hold for the full $\mathcal{L}\odot^{\mathcal{S}}$-logic, that contains the rule $\odot$-e, that enables to express measurement operators, which are not linear. Thus, our linearity theorem should concern the fragment of the $\mathcal{L}\odot^{\mathcal{S}}$-logic without this rule. But, if $\odot$-e rule is excluded, the connective $\odot$ is just the conjunction, and this fragment of the $\mathcal{L}\odot^{\mathcal{S}}$-logic is the $\mathcal{L}^{\mathcal{S}}$-logic.

So, for a greater generality, we prove our linearity theorem for the $\mathcal{L}^{\mathcal{S}}$-logic: linear logic with the interstitial rules and scalars, but without the $\odot$ connective, and discuss, at the end of the paper [6], how this result extends to the $\mathcal{L}\odot^{\mathcal{S}}$-logic.

Linear connectives. In the $\mathcal{L}^{\mathcal{S}}$-logic, we have to make a choice of connectives.

In intuitionistic linear logic, there is no multiplicative falsehood, no additive implication, and no multiplicative disjunction. Thus, we have two possible truths and two possible conjunctions, but only one possible falsehood, implication, and disjunction.

In the $\mathcal{L}^{\mathcal{S}}$-logic, we have chosen a multiplicative truth, an additive falsehood, a multiplicative implication, an additive conjunction, and an additive disjunction. The rule sum also is additive. The choice is not arbitrary, some variants do not exists in intuitionistic linear logic, others where necessary due to the interstitial rules.

Plan of the paper [6]. We first define the $\mathcal{L}^{\mathcal{S}}$-logic and its proof-language: the $\mathcal{L}^{\mathcal{S}}$-calculus, and prove that it verifies the subject reduction, confluence, termination, and introduction properties. We then show how the vectors of $\mathcal{S}^n$ can be expressed in this calculus and how the irreducible closed proofs of some propositions are equipped with a structure of vector space. We prove that all linear functions from $\mathcal{S}^m$ to $\mathcal{S}^n$ can be expressed as proofs of an implication between such propositions. We then prove the main result of this paper: that, conversely, all the proofs of implications between such propositions are linear. Finally, we show how this result extents to the proof language of the $\mathcal{L}\odot^{\mathcal{S}}$-logic and how this language is a quantum programming language.

References

[1] T. Altenkirch and J. Grattage. A functional quantum programming language. In *Proceedings of LICS 2005*, pages 249–258. IEEE, 2005.

[2] P. Arrighi and G. Dowek. Lineal: A linear-algebraic lambda-calculus. *Logical Methods in Computer Science*, 13(1), 2017.

[3] R. Blute. Hopf algebras and linear logic. *Mathematical Structures in Computer Science*, 6(2):189–217, 1996.

[4] B. Coecke and A. Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.

[5] A. Díaz-Caro and G. Dowek. A new connective in natural deduction, and its application to quantum computing. In A. Cerone and P. Csaba Ölveczky, editors, *Prooceedings of the International Colloquium on Theoretical Aspects of Computing*, volume 12819 of *Lecture Notes in Computer Science*, pages 175–193. Springer, 2021. Long version accessible at arXiv:2012.08994.

[6] A. Díaz-Caro and G. Dowek. Linear lambda-calculus is linear. In A. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2022.

[7] A. Díaz-Caro, M. Guillermo, A. Miquel, and B. Valiron. Realizability in the unitary sphere. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019)*, pages 1–13, 2019.

[8] A. Díaz-Caro, G. Dowek, and J.P. Rinaldi. Two linearities for quantum computing in the lambda calculus. *Biosystems*, 2019.

[9] Th. Ehrhard. On Köthe sequence spaces and linear logic. *Mathematical Structures in Computer Science*, 12(5):579–623, 2002.

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

[11] J.-Y. Girard. Coherent banach spaces: A continuous denotational semantics. *Theoretical Computer Science*, 227(1-2):275–297, 1999.

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

[13] L. Vaux. The algebraic lambda calculus. *Mathematical Structures in Computer Science*, 19(5):1029–1059, 2009.

[14] M. Zorzi. On quantum lambda calculi: a foundational perspective. *Mathematical Structures in Computer Science*, 26(7):1107–1195, 2016.

Linear Rank Intersection Types

Fábio Reis[1,2], Sandra Alves[1,2,3], and Mário Florido[1,2]

[1] DCC-FCUP, University of Porto, Porto, Portugal
[2] LIACC - Artificial Intelligence and Computer Science Laboratory
[3] CRACS, INESC-TEC - Centre for Research in Advanced Computing Systems

1 Intersection Types

Intersection type systems [2] characterize termination, in the sense that a λ-term is strongly-normalizable if and only if it is typable in an intersection type system. Thus, typability is undecidable for these systems.

To get around this, some current intersection type systems are restricted to types of finite rank [4, 8, 9, 12], using a notion of rank first defined by Daniel Leivant in [11]. This restriction makes type inference decidable [9]. Despite using finite-rank intersection types, these systems are still very powerful and useful. For instance, rank 2 intersection type systems [4, 8, 12] are more powerful, in the sense that they can type strictly more terms, than popular systems like the ML type system. Intersections arise in different systems in different scopes. Here we follow several previous presentations where intersections are only allowed in the left-hand side of arrow types [2, 3, 8, 13].

Definition 1 (Rank of intersection types)**.** Let $\mathbb{T}_0$ be the set of simple types and $\mathbb{T}_1 = \{\rho_1 \cap \cdots \cap \rho_m \mid \rho_1, \ldots, \rho_m \in \mathbb{T}_0, m \geq 1\}$. The set $\mathbb{T}_k$, of rank k intersection types (for $k \geq 2$), can be defined recursively in the following way ($n \geq 3$):

$$\mathbb{T}_2 = \mathbb{T}_0 \cup \{\sigma \to \tau \mid \sigma \in \mathbb{T}_1, \tau \in \mathbb{T}_2\}$$
$$\mathbb{T}_n = \mathbb{T}_{n-1} \cup \{\sigma_1 \cap \cdots \cap \sigma_m \to \tau \mid \sigma_1, \ldots, \sigma_m \in \mathbb{T}_{n-1}, \tau \in \mathbb{T}_n\}$$

Note that the rank of an intersection type is related to the depth of the nested intersections and it can be easily determined by examining it in tree form: a type is of rank k if no path from the root of the type to an intersection type constructor $\cap$ passes to the left of k arrows.

Quantitative types [1, 5, 6, 10] provide more than just qualitative information about programs and are particularly useful in contexts where we are interested in measuring the use of resources, as they are related to the consumption of time and space in programs. These systems are based on non-idempotent intersection types, using non-idempotence to count the number of evaluation steps and the size of the result.

In this work, we explore a type inference algorithm for non-idempotent rank 2 intersection types. Note that the set of terms typed using idempotent rank 2 intersection types and non-idempotent rank 2 intersection types is not the same. For instance, the term $(\lambda x.xx)(\lambda fx.f(fx))$ is typable with a simple type when using idempotent intersection types, but not when using non-idempotent intersection types. In fact, the only terms typable with a simple type in a non-idempotent intersection type system are the linear terms. This motivated us to come up with a new notion of rank for non-idempotent intersection types, based on linear types, that we here call *linear rank*.

2 Linear Rank

The relation between non-idempotent intersection types and linearity was introduced by Kfoury [10] and further explored by de Carvalho [5] who established its relation with linear logic.

Here we propose a new definition of rank for intersection types, which we call *linear rank* and differs from the previous one in the base case – instead of simple types, *linear rank* 0 intersection types are the linear types (the ones typed in a linear type system – a substructural type system in which each assumption must be used exactly once, corresponding to the implicational fragments of linear logic [7]).

Definition 2 (Linear rank of intersection types)**.** Let $\mathbb{T}_{\mathbb{L}0}$ be the set of linear types and $\mathbb{T}_{\mathbb{L}1} = \{\rho_1 \cap \cdots \cap \rho_m \mid \rho_1, \ldots, \rho_m \in \mathbb{T}_{\mathbb{L}0}, m \geq 1\}$. The set $\mathbb{T}_{\mathbb{L}k}$, of *linear rank* k intersection types (for $k \geq 2$), can be defined recursively in the following way ($n \geq 3$, $m \geq 2$):

$$\begin{aligned}
\mathbb{T}_{\mathbb{L}2} &= \mathbb{T}_{\mathbb{L}0} \cup \{\rho \multimap \tau \mid \rho \in \mathbb{T}_{\mathbb{L}0}, \tau \in \mathbb{T}_{\mathbb{L}2}\} \\
&\quad \cup \{\sigma_1 \cap \cdots \cap \sigma_m \to \tau \mid \sigma_1, \ldots, \sigma_m \in \mathbb{T}_{\mathbb{L}0}, \tau \in \mathbb{T}_{\mathbb{L}2}\} \\
\mathbb{T}_{\mathbb{L}n} &= \mathbb{T}_{\mathbb{L}n-1} \cup \{\sigma \multimap \tau \mid \sigma \in \mathbb{T}_{\mathbb{L}n-1}, \tau \in \mathbb{T}_{\mathbb{L}n}\} \\
&\quad \cup \{\sigma_1 \cap \cdots \cap \sigma_m \to \tau \mid \sigma_1, \ldots, \sigma_m \in \mathbb{T}_{\mathbb{L}n-1}, \tau \in \mathbb{T}_{\mathbb{L}n}\}
\end{aligned}$$

The idea for this change arose from our interest in using rank-restricted intersection types to estimate the number of evaluation steps of a λ-term while inferring its type. While defining the intersection type system to obtain quantitative information, we realized that the ranks could be potentially more useful for that matter if the base case was changed to types that give more quantitative information in comparison to simple types, which is the case for linear types – for instance, if a term is typed with a *linear rank* 2 intersection type, one knows that its arguments are linear, meaning that they will be used exactly once.

It is not clear, and most likely non-trivial, the relation between the standard definition of rank and our definition of *linear rank*, but it is an interesting question that we would like to explore in the future.

A glimpse of how differently we type linear and non-linear functions is given by the two arrow elimination rules of our Linear Rank 2 Intersection Type System:

$$\frac{\Gamma \vdash_2 M_1 : \rho_1 \cap \cdots \cap \rho_n \to \tau \qquad \Gamma_1 \vdash_2 M_2 : \rho_1 \quad \cdots \quad \Gamma_n \vdash_2 M_2 : \rho_n \qquad n \geq 2}{\Gamma, \sum_{i=1}^{n} \Gamma_i \vdash_2 M_1 M_2 : \tau} \quad (\to \text{ Elim})$$

$$\frac{\Gamma_1 \vdash_2 M_1 : \rho \multimap \tau \qquad \Gamma_2 \vdash_2 M_2 : \rho}{\Gamma_1, \Gamma_2 \vdash_2 M_1 M_2 : \tau} \quad (\multimap \text{ Elim})$$

Note that intersection is non-idempotent, thus when a term is typed by the ($\to$ Elim) rule, it is necessarily non-linear.

3 Contributions

The main contributions of this work are: a new notion of *linear rank*, a *linear rank* 2 non-idempotent intersection type system and a type inference algorithm which is sound and complete with respect to the type system. Our algorithm is inspired by previous type inference algorithms for rank 2 idempotent intersection types [8, 12]. Non-idempotent intersection types are quantitative types, thus we argue that our type inference algorithm is a first step towards the automatic inference of quantitative types.

Acknowledgments This work was financially supported within projects UIDB/00027/2020 and UIDB/50014/2020, funded by national funds through the FCT/MCTES (PIDDAC).

References

[1] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Log. J. IGPL*, 25(4):431–464, 2017.

[2] 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.

[3] Mario Coppo. An extended polymorphic type system for applicative languages. In Piotr Dembinski, editor, *Mathematical Foundations of Computer Science 1980 (MFCS'80), Proceedings of the 9th Symposium, Rydzyna, Poland, September 1-5, 1980*, volume 88 of *Lecture Notes in Computer Science*, pages 194–204. Springer, 1980.

[4] Ferruccio Damiani. Rank 2 intersection for recursive definitions. *Fundamenta Informaticae*, 77(4):451–488, 2007.

[5] Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. Phd thesis, Université Aix-Marseille II, 2007.

[6] Philippa Gardner. Discovering needed reductions using type theory. In *TACS*, volume 789 of *LNCS*, pages 555–574. Springer, 1994.

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

[8] Trevor Jim. Rank 2 type systems and recursive definitions. *Massachusetts Institute of Technology, Cambridge, MA*, 1995.

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

[10] Assaf Kfoury. A linearization of the lambda-calculus and consequences. *Journal of Logic and Computation*, 10(3):411–436, 2000.

[11] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 88–98, 1983.

[12] Steffen Van Bakel. *Intersection type disciplines in lambda calculus and applicative term rewriting systems*. Phd thesis, Mathematisch Centrum, Katholieke Universiteit Nijmegen, 1993.

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

Monsters: Programming and Reasoning

Venanzio Capretta[1] and Christopher Purdy[2]

[1] University of Nottingham, UK, venanzio.capretta@nottingham.ac.uk
[2] Cambridge University, UK, cp766@cam.ac.uk

A monadic stream (which we call a *monster*) is a potentially infinite sequence of values in which every element triggers a monadic action. Monsters are useful tools in functional programming: they can be instantiated to pure streams, lazy lists, finitely branching trees, interactive processes, state machines, and many other data structures.

A monster σ consists of an action for some monad M that, when executed, returns a head (first element) and a tail (continuation of the stream). This process is repeated in non-well-founded progression: monsters form a coinductive type.

Here is the formal type-theoretic definition of the set of streams with base monad M and elements of type A (M-*monsters*), in Haskell/Agda notation:

$$\begin{aligned} &\textbf{codata}\ \mathbb{S}_M\ A : \mathsf{Set} \\ &\quad \mathsf{mcons}_M : M\ (A \times \mathbb{S}_M\ A) \to \mathbb{S}_M\ A \end{aligned}$$

A previous article [4] introduced monadic streams and proved that polymorphic discrete functions on them are always continuous. A slightly different definition of monadic stream functions have been studied previously by Perez, Bärenz and Nilsson [7] to model signal processors. The definition of M-monsters is very close to that of *cofree (or iterative) comonad*, which can be seen as the type of M-monsters with a pure leading value [2, 5].

The functor M needs not be a monad for the type to be well-defined, though it enjoys some convenient properties when it is. For example, when M is a monad, monadic stream functions (isomorphic to monsters with the underlying functor ReaderT M) are arrows [7]. However, for the coinductive definition of $\mathbb{S}_M$ to be sound, the functor $M(A\times -)$ must have a final coalgebra. This is the case, for example, if M is a container [1]. (Monadic containers, in particular, are related to universes closed under Σ-types [3].)

Some important data structures are obtained as instances of monsters. If we choose M to be the identity functor, we obtain *pure streams*, that is, infinite sequences of elements of A. If we choose M to be the Maybe monad, we obtain *lazy lists*: the Just constructor returns a head element and a tail; the Nothing constructor terminates the list; since the type is coinductive, lists may go on forever. If we choose M to be the List constructor, we obtain *finitely branching trees*: a node consists of a list of children, each comprising an element of A and a subtree; if the list is empty we have a leaf; since the type is coinductive, trees need not be well-founded. If we choose M to be the State monad, we obtain *state machines*: processes that output an infinite stream of values depending on an underlying mutable state. If we choose M to be the IO monad, we obtain *interactive processes*: every stage of the stream is an input-output action that returns an element and a new process.

We developed an extensive library of generic functions for monsters in Haskell, publicly available on GitHub at https://github.com/venanzio/monster.
It provides generalizations of many operations on lists, streams, trees, and state machines. They allow high-level programming of abstract algorithms that can be instantiated to those data structures and others.

We also defined instances of the type classes of Functor/Applicative/Monad for $\mathbb{S}_M$. However, these satisfy the corresponding class laws only under certain conditions. If M is a functor,

it is straightforward to prove that $\mathbb{S}_M$ is a functor. If M is applicative, we proved that $\mathbb{S}_M$ is also applicative: it is surprisingly hard to establish this fact; the proof is complex and requires the definition of new operators and several intermediate technical lemmas. We are working on the verification of these results in Coq [9] and Agda [8]. We're exploring the possibility that a simpler proof could be derived from some abstract theorems on lax monoidal functors [6].

Finally, $\mathbb{S}_M$ is not in general a monad, even when M is: we showed this by a counterexample for State-monsters that violates the monadic laws. The monad class requires the definition of an operator $\mathsf{join} : \mathbb{S}_M\ (\mathbb{S}_M\ A) \to M\ A$ satisfying certain laws. We can see an element of $\mathbb{S}_M\ (\mathbb{S}_M\ A)$ as a *monster matrix*, a 2-dimensional array of elements of A in which both columns and rows emerge from M-actions. This must be compressed into a linear monster: it could only be done (generalizing the instantiation for pure streams) by travelling down the diagonal. However, there is no way, in a monster matrix, to step from one diagonal element to the next: we must always start from the outside of the matrix and choose the next column from there. Accordingly, the evaluation of each element on the diagonal activates the same M-actions repeatedly: this results in the failure of the monadic laws.

To ensure that $\mathbb{S}_M$ is a monad, M must satisfy additional requirements. We are looking for the minimal set of these, but we know it is sufficient for M to be a *representable* monad (which is the case for several common instances like Maybe and List).

In conclusion, we developed an extensive Haskell library on monadic streams (*monsters*) that provides many high-level operators that can be used on a wide range of important data structures. We also provide instances of the type classes Functor, Applicative and Monad, which hold valid under some assumption on the underlying type operator M.

References

[1] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[2] Peter Aczel, Jirí Adámek, Stefan Milius, and Jiri Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theor. Comput. Sci.*, 300(1-3):1–45, 2003.

[3] Thorsten Altenkirch and Gun Pinyo. Monadic containers and Σ-universes. In *TYPES 2017 Abstracts*, pages 20–21, 2017.

[4] Venanzio Capretta and Jonathan Fowler. The continuity of monadic stream functions. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[5] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Recursive coalgebras from comonads. *Information and Computation*, 204(4):437–468, April 2006.

[6] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[7] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 33–44. ACM, 2016.

[8] The Agda Team. *Agda User Manual*, 2021. Release 2.6.2.

[9] The Coq Development Team. *The Coq Proof Assistant. Reference Manual. Version 8.5*. INRIA, 2016. `http://coq.inria.fr/refman/index.html`.

More on modal embeddings and calling paradigms

José Espírito Santo[1], Luís Pinto[1], and Tarmo Uustalu[2,3]

[1] Centro de Matemática, Universidade do Minho, Portugal
[2] Dept. of Computer Science, Reykjavik University, Iceland
[3] Dept. of Software Science, Tallinn University of Technology, Estonia

The first connection between modal embeddings and calling paradigms in functional programming was made in the context of linear logic [7, 9]. In our work, we seek the root and the deep meaning of this connection, along the following general lines first laid out in [5]: (1) One can identify a modal calculus (a simple extension of the λ-calculus with an S4 modality) that serves as target of the modal embeddings and show that this modal target obeys a new calling paradigm, named call-by-box; (2) Two embeddings of intuitionistic logic into modal logic S4, as given in [11], where they are attributed to Girard and Gödel, can be recast as maps compiling respectively the ordinary (call-by-name, cbn) λ-calculus [1] and Plotkin's call-by-value (cbv) λ-calculus [10] into the modal target, both following the compilation technique of "protecting by a lambda", and together achieving a unification of call-by-name and call-by-value through call-by-box – all this provided Gödel's embedding suffers a slight adjustment, which, despite seeming innocent, allows a certain uniformity and simplicity in the images of the two embeddings, leading to the removal of all the β-reduction steps pertaining to the modality; (3) One can define later instantiations of the S4 modality, in the form of interpretations of the modal target into diverse other calculi, recovering by composition known embeddings like, for instance, the ones into the linear λ-calculus [9].

In the context of linear logic [9], one could already observe an asymmetry between the Girard/cbn embedding and Gödel/cbv one, with the latter enjoying slightly weaker properties. In our work [5], such an asymmetry remained: For cbn, the treatment is so neat that we may say Girard's embedding just points out an isomorphic copy of the cbn λ-calculus as a fragment of the modal target calculus. For cbv and Gödel's embedding, the results were not so satisfying. In a paper recently published [6], we investigate whether this asymmetry is inherent, or whether the modal analysis of the calling paradigms can be pushed further and reveal a hidden symmetry. We intend to present the results of this further investigation.

Recall that the modal target defined in [5] adds to the ordinary λ-calculus the term constructor $\mathsf{box}(M)$, while variables are written $\varepsilon(x)$ as a reminder of its special typing. Indeed, the type form $\Box A$ is added to simple types, contexts Γ only assign such modal types, $\mathsf{box}(M)$ has type $\Box A$ when M has type A, and $\varepsilon(x)$ has type A, if x is assigned $\Box A$. In addition, arrow types always have a modal type as antecedent. As to reduction, there is a single rule, for the contraction of β-redexes where the argument necessarily has the form $\mathsf{box}(M)$. A substitution is triggered, of M (typically of type $\Box A$) for some variable x of type A: this false mismatch is not real because there are, involved in this operation, implicit β-reduction steps of the kind engendered by the modality $\Box$.

The refinement of this modal system we propose in our new work consists in prohibiting nested modal types like $\Box\Box A$, and building into the syntax of terms T a minimum of typing information – namely the distinction between terms P, Q that can have a modal type and terms M, N that cannot. Abstractions $\lambda x.T$ and eliminations of the modality $\varepsilon(x)$ cannot have modal type, introductions of the modality $\mathsf{box}(M)$ have modal type, but applications MQ are ambiguous, so we separate two forms of them. Only now, after this separation, stemming from modal considerations alone, do we resort to an idea already employed in [5], namely the separation of a derived notion of reduction, concerning one of the forms of the application

constructor. These developments create in the target system two co-existing *modes*, the "left-first" and the "right-first", with which we can qualify the application constructor and reduction. The distinction between modes turns out to be connected to the distinction between calling paradigms. We verify that the modal embeddings are characterized by the mode they give preference to: Girard's embedding translates application and reduction to left-first application and reduction, whereas Gödel's embedding prefers the right-first mode.

By refining the modal target calculus in this way and accordingly recasting the embeddings, we find out that we do not lose the neat treatment of cbn, and in addition we obtain similarly neat results for cbv, that is: Gödel's embedding becomes just the indication of an isomorphic copy of Plotkin's cbv λ-calculus as a fragment of the refined modal target. In this sense, the ordinary and Plotkin's λ-calculi, more than being unified by the original modal target, they truly co-exist inside the refined (but still simple) modal calculus.

We plan to revisit the idea of instantiation of the modality from this new standpoint. We can return to the decomposition of the embeddings into the linear λ-calculus, briefly mentioned in [5] (in this regard, the study of the bang calculus [2, 3, 4] has somewhat related goals). But we can also consider embeddings into call-by-push-value [8] in the same spirit. The thesis we would like to confirm is that calculi encompassing call-by-name and call-by-value do so because they already embed our modal target.

References

[1] H. Barendregt, The Lambda Calculus, Vol. 103 of Studies in Logic and the Foundations of Mathematics, North-Holland, 1984.

[2] A. Bucciarelli, D. Kesner, A. Ríos, A. Viso, The bang calculus revisited, in: K. Nakano, K. Sagonas (Eds.), Functional and Logic Programming: 15th Int. Symp., FLOPS 2020, Proceedings, Vol. 12073 of Lecture Notes in Computer Science, Springer, 2020, pp. 13–32.

[3] T. Ehrhard, G. Guerrieri, The bang calculus: An untyped lambda-calculus generalizing call-by-name and call-by-value, in: Proc. of 18th Int. Symp. on Principles and Practice of Declarative Programming, PPDP '16, ACM, 2016, pp. 174–187.

[4] G. Guerrieri, G. Manzonetto, The bang calculus and the two Girard's translations, in: T. Ehrhard, M. Fernández, V. de Paiva, L. T. de Falco (Eds.), Proc. of Joint Int. Workshops on Linearity and Trends in Linear Logic and Applications, Linearity-TLLA 2018, Vol. 292 of Electronic Proceedings in Theoretical Computer Science, Open Publishing Assoc., 2019, pp. 15–30.

[5] J. Espírito Santo, L. Pinto, T. Uustalu, Modal embeddings and calling paradigms, in: H. Geuvers (Ed.), 4th Int. Conf. on Formal Structures for Computation and Deduction, FSCD 2019, Vol. 131 of Leibniz Int. Proc. in Informatics, Dagstuhl Publishing, 2019, pp. 18:1–18:20.

[6] J. Espírito Santo, L. Pinto, T. Uustalu, Plotkin's call-by-value λ-calculus as a modal calculus, Journal of Logical and Algebraic Methods in Programming 127 (2022) 100775

[7] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1) (1987) 1–102.

[8] P. B. Levy, Call-by-push-value: Decomposing call-by-value and call-by-name, Higher-Order and Symbolic Compututation 19 (4) (2006) 377–414.

[9] J. Maraist, M. Odersky, D. N. Turner, P. Wadler, Call-by-name, call-by-value, call-by-need and the linear lambda calculus, Theoretical Computer Science 228 (1–2) (1999) 175–210.

[10] G. Plotkin, Call-by-name, call-by-value and the λ-calculus, Theoretical Computer Science 1 (1975) 125–159.

[11] A. Troelstra, H. Schwichtenberg, Basic Proof Theory, 2nd Edition, Vol. 43 of Cambridge Tracts in Theoretical Computer Science, Cambridge Univ. Press, 2000.

Myhill Isomorphism Theorem and a Computational Cantor-Bernstein Theorem in Constructive Type Theory

Yannick Forster[1,2], Felix Jahn[1], and Gert Smolka[1]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] Inria, Gallinette Project-Team, Nantes, France

Abstract

Two enumerable discrete types which can be embedded into each other in constructive type theory are isomorphic. Furthermore, the isomomorphism can be constructed to preserve a reduction property regarding predicates on the types. This (novel) result can be interpreted in two ways: First, it is a computational form of the Cantor-Bernstein theorem [2, 1] from set theory, stating that two sets which can be embedded into each other are in bijection, which is is inherently classical in its full generality [8]. Secondly, it is a synthetic form of the Myhill isomorphism theorem [7] from computability theory stating that 1-equivalent predicates are recursively isomorphic.

We give two proofs of the result: The first is following the textbook proof of the Myhill isomorphism theorem closely, and thus preserves reduction of predicates. The second is a direct proof, which seems to not be extendable to predicates. All proofs are machine-checked in Coq but should transport to other foundations – they do not rely on impredicativity, on choice principles, or on large eliminations apart from falsity elimination.

A type X is enumerable and discrete if and only if it is a retract of $\mathbb{N}$ (see e.g. [4, Corr. 4.34]), i.e. if there are $I\colon X\to\mathbb{N}$ and $R\colon\mathbb{N}\to\mathbb{O}X$ with $\forall x.\ R(Ix) = \mathsf{Some}\ x$.

1 Myhill isomorphism theorem

A predicate $p\colon X\to\mathbb{P}$ is one-one reducible to a predicate $q\colon Y\to\mathbb{P}$ if $p \preceq_1 q := \exists f\colon X\to Y.\ (\forall x.\ px \leftrightarrow q(fx)) \wedge f$ is injective. We prove that when two predicates are one-one reducible to each other, where the reductions functions are in no relation, one can construct one-one reductions which also invert each other. We follow Rogers [9, §7.4 Th. VI], where the isomorphism is constructed in stages, formed by *correspondence sequences* between predicates p and q, which are finitary bijections represented as lists $C\colon\mathbb{L}(X\times Y)$ such that for all $(x,y)\in C$

$$1.\ px\leftrightarrow qy \qquad 2.\ \forall y'.\ (x,y')\in C\to y=y' \qquad 3.\ \forall x'.\ (x',y)\in C\to x=x'$$

We write $x\in_1 C$ ($x\in_2 C$) if x is an element of the first (second) projection of C.

The crux of the theorem is that for any correspondence sequence C with $p\preceq_1 q$ and $x_0\notin_1 C$ one can *compute* y_0 such that $(x_0,y_0)::C$ is a correspondence sequence again.

Lemma 1. *Let f be a one-one reduction from p to q. There is a function* $\mathsf{find}\colon\mathbb{L}(X\times Y)\to X\to Y$ *such that if C is a correspondence sequence for p and q and $x_0\notin_1 C$, then* $\mathsf{find}\ C\ x_0\notin_2 C$ *and* $px_0\leftrightarrow q(\mathsf{find}\ C\ x_0)$.

Proof. We first define a function $\gamma\colon\mathbb{L}(X\times Y)\to X\to X$ recursive in $|C|$:

$$\gamma Cx := x \ \text{ if } fx\notin_2 C \qquad \gamma Cx := \gamma\,(\mathsf{filter}(\lambda t.t\neq_{\mathbb{B}}(x',fx))\,C)\,x' \ \text{ if } (x',fx)\in C$$

For a correspondence sequence C between p and q and $x\notin_1 C$ we have (1) $px\leftrightarrow p(\gamma Cx)$, (2) $\gamma Cx = x$ or $\gamma Cx\in_1 C$, and (3) $f(\gamma Cx)\notin_2 C$. The proof is by induction on the length of C, exploiting the injectivity of f. Now $\mathsf{find}\ Cx_0 := f(\gamma Cx_0)$ is the wanted function. □

For the rest of this section we fix enumerable discrete types X and Y such that (I_X, R_X) and (I_Y, R_Y) are retractions from X and Y respectively to $\mathbb{N}$. We construct the isomorphism via a cumulative correspondence sequence C_n with $I_X x < n \to x \in_1 C_n$ and $I_Y y < n \to y \in_2 C_n$.

$$C'_n := \begin{cases} (x, \mathsf{find}\, C_n\, x) :: C_n \text{ if } R_X n = \mathsf{Some}\, x \wedge x \notin_1 C_n \\ C_n \text{ otherwise} \end{cases} \quad C_{n+1} := \begin{cases} (\mathsf{find}\, \overleftrightarrow{C'_n}\, y, y) :: C'_n \text{ if } R_Y n = \mathsf{Some}\, y \wedge y \notin_2 C'_n \\ C'_n \text{ otherwise} \end{cases}$$

where $C_0 := []$ and $\overleftrightarrow{C} := \mathsf{map}\,(\lambda(x,y).(y,x))\,C$.

Lemma 2. *C_n is a correspondence sequence for p and q such that*

$$1.\ n \le m \to C_n \subseteq C_m \qquad 2.\ I_X x < n \to x \in_1 C_n \qquad 3.\ I_Y y < n \to y \in_2 C_n$$

Theorem 1 (Myhill)**.** *Let X and Y be enumerable discrete types, $p\colon X \to \mathbb{P}$, and $q\colon Y \to \mathbb{P}$. If $p \preceq_1 q$ and $q \preceq_1 p$, then there exist $f\colon X \to Y$ and $g\colon Y \to X$ such that for all $x\colon X$ and $y\colon Y$:*

$$px \leftrightarrow q(fx), \quad qy \leftrightarrow p(gy), \quad g(fx) = x, \quad f(gy) = y$$

Proof. fx is defined as the unique y for which $(x, y) \in C_{I_X x+1}$ (which exists by Lemma 2 (2) and is unique because $C_{I_X x+1}$ is a correpsondence sequence), and gy is symmetrically defined as the unique x for which $(x, y) \in C_{I_Y y+1}$. (1) and (2) are immediate since C_n is a correspondence sequence. (3) and (4) are by case analysis whether $I_X x \le I_Y y$ or vice versa. □

Proofs of the theorem have appeared in different forms in [6], [4], and [5].

2 Computational Cantor-Bernstein Theorem

The Cantor-Bernstein theorem is inherently classical in set theory [8]. Recently, the classical proof has been generalised to all boolean toposes by Escardó [3]. We give a direct, fully constructive proof of the Cantor-Bernstein theorem for enumerable discrete types via a novel alignment theorem. We say that a type X is aligned if there are $A\colon X \to \mathbb{N}$ and $B\colon X \to \mathbb{N} \to \mathbb{O}X$ s.t. $\forall x. \forall n \le Ax. \exists y.\ Bxn = \mathsf{Some}\, y \wedge Ay = n$. We write $\#l$ if a list l is duplicate-free.

Theorem 2 (Alignment)**.** *Every enumerable discrete type is aligned.*

Proof. Let (I, R) be a retraction from X to $\mathbb{N}$. We define Ax to be the position of x in L_{Ix}, where L_n is the list of all x s.t. $\exists m \le n.\ Rm = \mathsf{Some}\, x$. Bxn is the n-th element of L_{Ix} if it exists, and x otherwise. □

Lemma 3. *Let X be aligned by $s\colon X \to \mathbb{N}$.*

1. *There is $F'_X\colon \mathbb{N} \to X \to \mathbb{L}X \to X$ s.t. if $\#l$ and $|l| = n + 1$ then $s(F'_X x_0 n l) = n$.*
2. *There is $F_X\colon X \to \mathbb{L}X$ s.t. $\#(F_X x)$ and $|F_X x| = 1 + sx$.*

Proof. (1) uses a function $\mathsf{get}\colon \mathbb{L}\mathbb{N} \to \mathbb{N}$ s.t. if $\#l$ and $|l| = n+1$, then $\mathsf{get}\ l \in l$ and $\mathsf{get}\ l \ge n$. □

Lemma 4. *Aligned types X, Y with injections $X \to Y$ and $Y \to X$ are isomorphic.*

Proof. Let X and Y be aligned and $f\colon X \to Y$ and $g\colon Y \to X$ be injections. Define $f'x := F'_Y(A_X x)(fx)(\mathsf{map}\ f\ (F_X x))$ and $g'y := F'_X(A_Y y)(gy)(\mathsf{map}\ g\ (F_Y y))$. We have that $\#(\mathsf{map}\ f\ (F_X x))$, which follows because f is injective and $\#(F_X x)$, and $|\mathsf{map}\ f\ (F_X x)| = s_X x + 1$, which is immediate. This suffices with the dual properties for $\mathsf{map}\ g\ (F_Y y)$. □

Theorem 3 (Computational Cantor-Bernstein)**.** *For enumerable discrete types X, Y with injections $X \to Y$ and $Y \to X$ there are functions $X \to Y$ and $Y \to X$ inverting each other.*

Proof. Either by applying the preceding lemma and the alignment theorem, or by applying the Myhill isomorphism theorem with $px := \top$ and $qy := \top$. □

References

[1] Felix Bernstein. Untersuchungen aus der mengenlehre. *Mathematische Annalen*, 61(1):117–155, March 1905. doi:10.1007/bf01457734.

[2] Georg Cantor. Mitteilungen zur Lehre vom Transfiniten. In *Zeitschrift für Philosophie und philosophische Kritik 91*, 1887.

[3] Martín Hötzel Escardó. The cantor–schröder–bernstein theorem for $$\infty $$-groupoids. *Journal of Homotopy and Related Structures*, 16(3):363–366, June 2021. doi:10.1007/s40062-021-00284-6.

[4] Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021. URL: https://ps.uni-saarland.de/~forster/thesis.

[5] Yannick Forster, Felix Jahn, and Gert Smolka. A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq (Full Version). preprint, February 2022. URL: https://hal.inria.fr/hal-03580081.

[6] Felix Jahn. *Synthetic One-One, Many-One, and Truth-Table Reducibility in Coq*. Bachelor's thesis, Saarland University, 2020. URL: https://ps.uni-saarland.de/~jahn/bachelor.php.

[7] John Myhill. Creative sets. 1957. doi:10.1002/malq.19550010205.

[8] Pierre Pradic and Chad E. Brown. Cantor-Bernstein implies excluded middle. *CoRR*, abs/1904.09193, 2019. arXiv:1904.09193.

[9] Hartley Rogers. Theory of recursive functions and effective computability. 1987.

Noninvasive Polarized Subtyping for Inductive Types

Théo Laurent[1] and Kenji Maillard[2]

[1] Inria Paris, Prosecco Team
[2] Inria Rennes-Bretagne Atlantique, Gallinette Team

Abstract

Polarized subtyping extends a type system with annotations tracking the variance of type constructors with respect to subtyping. Type system with polarized subtyping are highly expressive at the cost of additional information decorating types and contexts. In the dependently typed setting, for instance inside proof assistants, this pervasive complexity obstruct its integration within existing implementations. We advocate a less invasive approach, extending the Calculus of Inductive Constructions (CIC) with a schema for polarized inductive types while retaining vanilla typing jugdments for most of the theory. The design of our extension is guided by a model exhibiting subtyping as a particular order structure on types.

Type Theory and Polarized Subtyping

The Calculus of Inductive Constructions (CIC) is both a powerful logic, used as the basis for many proof assistants, e.g. Coq, and an expressive type system underlying programming languages such as F* [9]. Dependent types, introduced notably though dependent product types $\Pi(x : A).\, B$ and a hierarchy of universes $\square_{i\in\mathbb{N}}$, provide a uniform language for programs as well as specifications. In practice, implementations of CIC sometimes extend the theory with different forms of subtyping to make it more flexible, for instance cumulativity in Coq [10] or refinement types in F*. Both of these instances however lack from structural forms of subtyping [5] for general inductive types. A structural subtyping discipline allows one to derive subtyping relationships between inductive types from their structure and assumptions on their parameters. For instance, in such a system one can derive that $\texttt{List}\, A$ is a subtype of $\texttt{List}\, B$ whenever A is a subtype of B. Said otherwise, $\texttt{List} : \square_i \to \square_i$ is monotone for the subtyping order endowing $\square_i$. *Polarities* [1, 7] endow types with annotations that capture their monotonicity. As an example, the function type constructor $(\to)$ can be given the annotated type $\Pi(A :_- \square_i).\Pi(B :_+ \square_i).\, \square_i$ expressing that it is antitone in its domain A and monotone in its codomain B with respect to subtyping. In a dependently typed setting, tracking these polarities pervasively in types and terms result in complex type theories that seems unlikely to be practical.

External Polarities for Inductive Types

We advocate for a pragmatic approach suitable for adoption by proof assistants. In order to stay closer to CIC, we restrict polarities to where they are most relevant in practice: on parameters of inductive types. This restriction allow us to keep polarities as *external* entities of the type theory. Following the practice of [8], we propose to collect inductive declarations in a signature Σ and only alter context and types so that they can be decorated with additional variance information. Our system feature four kind of variance: discrete (`dis`), covariant $(+)$, contravariant $(-)$ and codiscrete (`cod`). Continuing with the example of parametrized Lists, this inductive is represented as the following entry in the signature:

$$\Sigma_{\texttt{List}} := \{\, \texttt{List}\,(A :_+ \square_i) := \texttt{nil} \mid \texttt{cons}\,(hd : A, tl : \texttt{List}\, A)\,\}.$$

The parameter $A : \square_i$ is annotated with a polarity $+$ indicating that the type constructor `List` is monotonic with regard to subtyping in this parameter. For an inductive entry to be well formed with respect to its polarity annotations, we must ensure that the parameters are used adequately in the constructors. For `List`, the two instances of A in `cons` are indeed in positive positions. For general inductives, we use polarized typing judgments inspired by directed type theories [4, 7] to track adequately these polarities and formulate the well-formedness conditions.

The monotonicity informations captured through polarities in the signature induce corresponding structural subtyping rules and coercions [6]. In the case of `List`, this pschema generate the following rule:

$$\frac{\Sigma_{\texttt{List}}; \Gamma \vdash \texttt{List} : \Pi(A :_+ \square_i).\, \square_i \qquad \Sigma_{\texttt{List}}; \Gamma \vdash A \preceq B}{\Sigma_{\texttt{List}}; \Gamma \vdash \texttt{List}\, A \preceq \texttt{List}\, B} \tag{1}$$

Backing Up our Approach with a Syntactic Translation

To support the design of our type theory and establish its metatheory, we build upon the syntactic models of [3]. We construct a model for our theory by compiling it to Coq, translating a typing judgement $\Gamma \vdash t : A$ to $[\![\Gamma]\!] \vdash [t] : [\![A]\!]$. A peculiarity of our model is the use of a second translation $[\![-]\!]_\varepsilon$ for the interpretation of subtyping judgements. This translation equips any closed type A with a binary relation $[\![A]\!]_\varepsilon : [\![A]\!] \to [\![A]\!] \to \square_i$ in a fashion reminiscent of binary parametricity [2]. In particular, the subtyping order arises as the relation $[\![\square_i]\!]_\varepsilon$ on the universe. The stratification of the model into two distinct interpretations illustrate the external nature of polarities: indeed, $[\![-]\!]$ merely erases polarities and only $[\![-]\!]_\varepsilon$ take polarities into account.

The pair of translations $[-], [\![-]\!]$ is relatively simple, preserving all term and type construction from CIC homomorphically but universes $\square_i$ and subtyping coercions. Indeed, in order to maintain a stratification between typing and subtyping translations, we interpret subtyping coercions as functions defined by case analysis on types. We do so using universes of codes to reify types, following the approach of ad-hoc polymorphism in [3]. As a consequence, these coercions can be computed by structural induction on these codes, only relying on the mere existence of subtyping relations to discard impossible cases.

Conversely, $[-]_\varepsilon, [\![-]\!]_\varepsilon$ crucially uses the polarity-annotated types in the signature to ensure that the interpretation of type constructors are adequately monotone. In particular, the structural subtyping rules attached to each inductive is validated by this translation. Continuing with the example of `List`, the translation give us

$$\begin{aligned}
[\![\Pi(A :_+ \square_i).\, \square_i]\!]_\varepsilon\, [\texttt{List}]\ [\texttt{List}] &:= \Pi(A_1 : [\![\square_i]\!])(A_2 : [\![\square_i]\!]) \\
&\quad (A_\varepsilon : [\![\square_i]\!]_\varepsilon\, A_1\, A_2).\, [\![\square_i]\!]_\varepsilon\, ([\texttt{List}]\, A_1)\, ([\texttt{List}]\, A_2) \\
[\![A \preceq B]\!] &:= [\![\square_i]\!]_\varepsilon\, A\, B \\
[\![\texttt{List}\, A \preceq \texttt{List}\, B]\!] &:= [\![\square_i]\!]_\varepsilon\, ([\texttt{List}]\, A)\, ([\texttt{List}]\, B)
\end{aligned}$$

Hence, from the term $[\texttt{List}]_\varepsilon : [\![\Pi(A :_+ \square_i).\, \square_i]\!]_\varepsilon\, [\texttt{List}]\ [\texttt{List}]$ and a witness $[A \preceq B] : [\![A \preceq B]\!]$ we are able to construct $[\texttt{List}]_\varepsilon\, [A]\, [B]\, [A \preceq B] : [\![\texttt{List}\, A \preceq \texttt{List}\, B]\!]$, validating the rule (1).

References

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

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

[3] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 182–194, 2017.

[4] Daniel R. Licata and Robert Harper. 2-dimensional directed type theory. In *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011*, pages 263–289, 2011.

[5] Zhaohui Luo and Robin Adams. Structural subtyping for inductive types with functorial equality rules. *Math. Struct. Comput. Sci.*, 18(5):931–972, 2008.

[6] Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: Theory and implementation. *Inf. Comput.*, 223:18–42, 2013.

[7] Andreas Nuyts. Toward a directed homotopy type theory based on 4 kinds of variance. Master's thesis, Katholieke Universiteit Leuven, 2015.

[8] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The metacoq project. *J. Autom. Reason.*, 64(5):947–999, 2020.

[9] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[10] Amin Timany and Matthieu Sozeau. Cumulative inductive types in Coq. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 29:1–29:16, 2018.

On Dynamic Lifting and Effect Typing in Circuit Description Languages (Extended Abstract)

Andrea Colledan[1,2] and Ugo Dal Lago[1,2]

[1] University of Bologna, Italy
[2] INRIA Sophia Antipolis, France

Abstract

Despite the undeniable fact that large-scale, error-free quantum hardware has yet to be built [12], research into programming languages specifically designed to target architectures including quantum hardware has taken hold in recent years [11]. Most of the proposals in this sense (see [5, 15, 16] for some surveys) concern languages that either express or can be compiled into some form of *quantum circuit* [9], which can then be executed by quantum hardware. This reflects the need to have tighter control over the quantum resources that programs employ. In this scenario, the idea of considering high-level languages that are specifically designed to *describe* circuits and in which the latter are in fact first-class citizens is particularly appealing.

A typical example of this class of languages is `Quipper` [6, 7], whose underlying design principle is that of enriching a very expressive functional language like `Haskell` with the possibility of manipulating quantum circuits. In other words, programs do not just build circuits, but also treat them like data. `Quipper`'s meta-theory has been studied in recent years through the introduction of a family of research languages that correspond to suitable `Quipper` fragments and extensions, which usually take the form of linear λ-calculi. This family includes languages such as Proto-Quipper-S [14], Proto-Quipper-M [13], Proto-Quipper-D [3, 4] and Proto-Quipper-L [8].

An aspect that so far has only marginally been considered by the research community is the study of the meta-theory of so-called *dynamic lifting*, i.e. the possibility of allowing any classical value flowing in one of the wires of the underlying circuit, naturally unknown at circuit building time, to be *visible* in the host program for control flow. As an example, one could append a unitary to some wires *only if* a previously performed measurement has yielded a certain outcome. This is commonly achieved in many quantum algorithms via classical control, but `Quipper` also offers a higher-level solution precisely in the form of dynamic lifting, as shown in the example program in Figure 1. Notably, such a program cannot be captured by any of the calculi in the Proto-Quipper family, with the exception of Lee et al.'s Proto-Quipper-L [8], arguably the most recent addition to the family.

Looking at the `Quipper` program in Figure 1, one immediately realizes that the two branches of both occurrences of the `if` operator change the underlying circuit in a uniform way, i.e. the number and type of the wires are the same in either branch. What if, for instance, we wanted to condition the execution of a *measurement* on a lifted value, like in Figure 2? Such situations can arise, for example, in one-way quantum computing, and the so-called measurement calculus [2] indeed allows the execution of a measurement to be conditioned on the result of a previous measurement. Unfortunately, `Quipper` does *not* allow the program in Figure 2 to be typed. It is therefore natural to wonder whether this is an intrinsic limitation, or if a richer type system can deal with a more general form of circuits.

In this talk, we introduce a generalization of Proto-Quipper-M, called Proto-Quipper-K, in which dynamic lifting is available in a very general form. The evaluation of a Proto-Quipper-K term M can involve the lifting of a bit value into a variable u and consequently produce in

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b q a = do
 a <- qnot a `controlled` q
 q <- hadamard q
 (x,y) <- measure (q,a)
 (u,s) <- dynamic_lift (x,y)
 b <- if s then gate_X b else return b
 if u then gate_Z b else return b
```

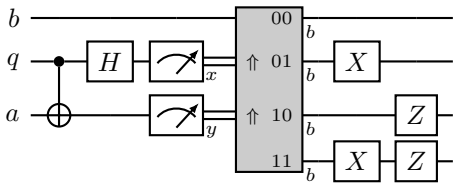


Figure 1: Quantum teleportation circuit with dynamic lifting. The gray box represents the dynamic lifting of bit wires x, y and the extension of the circuit on wire b with one of four possible continuations, depending on the results of the intermediate measurement.

```
oneWay ::  Qubit -> Qubit -> ???
oneWay q a = do
 q <- hadamard q
 x <- measure q
 u <- dynamic_lift x
 if u then measure a else return a
```

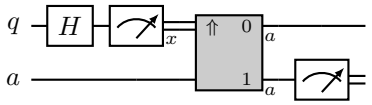


Figure 2: An example of conditional measurement. This program is ill-typed in `Quipper`.

output not a single result in the set VAL of values, but possibly one distinct result for each possible value of u. Therefore, it is natural to think of M as a computation that results in an object in the set $\mathcal{K}_{\{u\}}(VAL)$, where $\mathcal{K}_{\{u\}} = X \mapsto (\{u\} \to \{0,1\}) \to X$ is a functor such that an element of $\mathcal{K}_{\{u\}}(X)$ returns an element of X for each possible Boolean assignment to u.

This approach, though apparently straightforward, scales up in a nontrivial way. Since the lifting of a bit can happen conditionally on the value of a previously lifted variable, in the event that *more than one* variable is lifted, we have to resort to a tree-like structure to keep track of the eventual dependencies between consecutive liftings. We call such a structure a *lifting tree* and employ it pervasively throughout our work. In general, a program whose lifting pattern is captured by a lifting tree $\mathfrak{t}$ produces a *lifted value* as a result, namely an element of $\mathcal{K}_{\mathfrak{t}}(VAL)$, where $\mathcal{K}_{\mathfrak{t}} = X \mapsto (\mathscr{P}_{\mathfrak{t}} \to X)$ and $\mathscr{P}_{\mathfrak{t}}$ is the set of all assignments of lifted variables which describe valid root-to-leaf paths in $\mathfrak{t}$. Since by design we want to handle situations in which a circuit, and by necessity the term building it, can produce results which have distinct *types*—and not only distinct *values*—depending on the values of the lifted variables, we also employ (in the spirit of the type and effects paradigm [10]) an effectful notion of *type*, typing computations according to some *lifted type*, that is, an element of $\mathcal{K}_{\mathfrak{t}}(TYPE)$, where $TYPE$ is the set of Proto-Quipper-K types.

Proto-Quipper-K is thus capable of producing not only circuits like the one in Figure 1, but rather a more general class of circuits whose structure and type *essentially depend* on the values flowing through the lifted channels, like the one in Figure 2. The main results that we give, beside the introduction of the language itself, its type system, and its operational semantics, are the type soundness results of subject reduction and progress, which together let us conclude that well-typed Proto-Quipper-K programs do not go wrong from an operational point of view. An extended version of our work is already available in [1].

References

[1] Andrea Colledan and Ugo Dal Lago. On dynamic lifting and effect typing in circuit description languages (extended version), 2022.

[2] Vincent Danos, Elham Kashefi, and Prakash Panangaden. The measurement calculus. *J. ACM*, 54(2), April 2007.

[3] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. A tutorial introduction to quantum circuit programming in dependently typed proto-quipper. In Ivan Lanese and Mariusz Rawski, editors, *Proc. of RC*, pages 153–168, Cham, 2020.

[4] Peng Fu, Kohei Kishida, and Peter Selinger. Linear dependent type theory for quantum programming languages: Extended abstract. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *Proc. of LICS*, pages 440–453, 2020.

[5] Simon J. Gay. Quantum programming languages: Survey and bibliography. *Math. Struct. Comput. Sci.*, 16(4):581–600, August 2006.

[6] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in quipper. In *Proc. of RC*, pages 110–124, 2013.

[7] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper. In *Proc. of PLDI*, page 333–342, June 2013.

[8] Dongho Lee, Valentin Perrelle, Benoît Valiron, and Zhaowei Xu. Concrete Categorical Model of a Quantum Circuit Description Language with Measurement. In *Proc. of FSTTCS*, volume 213 of *LIPIcs*, pages 51:1–51:20, 2021.

[9] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[10] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design: Recent Insights and Advances*, pages 114–136. Springer Berlin Heidelberg, 1999.

[11] Jens Palsberg. Toward a universal quantum programming language. *XRDS: Crossroads*, 26(1):14–17, September 2019.

[12] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[13] Francisco Rios and Peter Selinger. A categorical model for a quantum circuit description language. In *Proc. of QPL*, pages 164–178, June 2017.

[14] Neil Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, 2015.

[15] Peter Selinger. A brief survey of quantum programming languages. In *Proc. of FLOPS*, pages 1–6, 2004.

[16] Mingsheng Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016.

Partial Dijkstra Monads for All

Théo Winterhalter[1], Cezar-Constantin Andrici[1], Cătălin Hriţcu[1],
Kenji Maillard[2], Guido Martínez[3], and Exequiel Rivas[4]

[1] MPI-SP [2] Inria Rennes [3] CIFASIS-CONICET and UNR Argentina [4] TUT

Abstract

Dijkstra Monads for All introduces a generic method to construct a Dijkstra monad from a monad morphism between a computation and a specification monad. However, applying this construction to usual computation monads yields Dijkstra monads that do not support partiality, which makes them unusable in F*. We show that this issue can be overcome when the computation and specification monads support partiality by providing a way to *require* pre-conditions, and we provide several techniques to build such monads.

Dijkstra monads are indexed monad structures that are used in F* for verifying effectful programs [SHK+16, SWS+13]. Concretely, a Dijkstra monad $D\ A\ w$ represents an effectful computation returning values of type A and obeying specification $w : W\ A$, where W is a specification monad. For instance, the state Dijkstra monad ST is usually specified by the monad $\mathsf{W}^{\mathsf{ST}}\ A = (A \times S \to \mathbb{P}) \to (S \to \mathbb{P})$, where $\mathbb{P}$ is the type of propositions. W^{ST} is the type of a predicate transformer taking a post-condition on the final state and a result value and returning a pre-condition on the initial state. We have the following Dijkstra monad interface:

$$\begin{array}{lcl@{\qquad}lcl} \mathbf{return}^{\mathsf{ST}}\ (x:A) & : & \mathsf{ST}\ A\ (\mathbf{return}^{\mathsf{W}}\ x) & \mathbf{return}^{\mathsf{W}} & = & \lambda p\ s_0.\ p\ (x, s_0) \\ \mathbf{get}^{\mathsf{ST}}\ () & : & \mathsf{ST}\ S\ \mathbf{get}^{\mathsf{W}} & \mathbf{get}^{\mathsf{W}} & = & \lambda p\ s_0.\ p\ (s_0, s_0) \\ \mathbf{put}^{\mathsf{ST}}\ (s:S) & : & \mathsf{ST}\ \mathbf{unit}\ (\mathbf{put}^{\mathsf{W}}\ s) & \mathbf{put}^{\mathsf{W}} & = & \lambda p\ s_0.\ p\ ((), s) \end{array}$$

$$\mathbf{bind}^{\mathsf{ST}}\ (c : \mathsf{ST}\ A\ w_c)\ (f : (x:A) \to \mathsf{ST}\ B\ (w_f\ x)) : \mathsf{ST}\ B\ (\mathbf{bind}^{\mathsf{W}}\ w_c\ w_f)$$
$$\mathbf{bind}^{\mathsf{W}}\ w_c\ w_f = \lambda p\ s_0.\ w_c\ (\lambda(x, s_1).\ w_f\ x\ p\ s_1)\ s_0$$

If we take a post-condition $p : A \times S \to \mathbb{P}$, we say it holds on program $\mathbf{return}^{\mathsf{ST}}\ x$ if we can prove $\mathbf{return}^{\mathsf{W}}\ x\ p$ on the initial state s_0 or in other words if $p\ (x, s_0)$ holds. For p to hold on $\mathbf{get}^{\mathsf{ST}}\ ()$ then it must hold on return value and final state both equal to the initial state: $p\ (s_0, s_0)$. For p to hold on $\mathbf{put}^{\mathsf{ST}}\ s$ it must hold on final state s and trivial unit value $()$: $p\ ((), s)$; the initial state is erased so it is ignored. Such typed Dijkstra monad interfaces allow F* to compute verification conditions simply by dependent type inference.

Constructing Dijkstra monads. *Dijkstra Monads for All* (DM4All) [MAA+19] introduces a generic way to construct Dijkstra monads. For any computation monad M, and for any ordered specification monad W with order $\leq^{\mathsf{W}}$, if there is a monad morphism $\theta : M \to W$ then one can define the following Dijkstra monad:

$$D\ A\ w = \{c : M\ A \mid \theta\ c \leq^{\mathsf{W}} w\} \tag{1}$$

For instance, from the usual state computation monad $\mathsf{State}\ A = S \to A \times S$ to the W^{ST} specification monad above one can define the monad morphism $\theta\ c = \lambda p\ s_0.\ p\ (c\ s_0)$. Another example is non-determinism, where we can take $M\ A := \mathbf{list}\ A$ as computation monad, $W\ A = (A \to \mathbb{P}) \to \mathbb{P}$ as specification monad, and $\theta^{\forall}\ c = \lambda p.\ (\forall x \in c.\ p\ x)$ as monad morphism, essentially saying that any post-condition should hold for all values stored in the list, or in other words for every possible outcome of the computation. This gives a demonic

interpretation of non-determinism, and one can also chose an angelic interpretation by using $\theta^{\exists}\ c = \lambda p.\ (\exists x \in c.\ p\ x)$ instead. This construction of Dijkstra monads neatly separates the syntax (M) from the specification (W) and semantics (θ) as one can *forget* the refinement and extract the value in $M\ A$ from $D\ A\ w$. While very general, the DM4All construction often produces Dijkstra monads that do not support partiality on standard computational monads, which makes them unusable in F*, as explained below.

F* and the partiality effect. The PURE effect (i.e., Dijkstra monad) of F* represents in fact *partial* computations, as for instance one can use the pre-condition to discard provably unreachable branches of a pattern-matching, and recursive functions can loop on arguments not satisfying the pre-condition. We can model this notion of partiality in a more standard dependent type theory via a require construct with the following type:

$$\textbf{require}\ (p : \mathbb{P}) : \mathsf{PURE}\ p\ (\lambda q.\ (\exists (h : p).\ q\ h))$$

It returns a proof of the proposition p that can then be used by the continuation. The specification requires p as a pre-condition (the $\exists(h : p)$ part) and also asks for the post-condition (q) to hold on the proof of p. We argue that the existence of such an operator is tantamount to supporting partiality. Concretely, we will say that a monad M supports partiality when there is $\textbf{require}^{\mathsf{M}}\ (p : \mathbb{P}) : M\ p$ and that a Dijkstra monad D supports partiality when its specification monad does too and we have $\textbf{require}^{\mathsf{D}}\ (p : \mathbb{P}) : D\ p\ (\textbf{require}^{\mathsf{W}}\ p)$.

In F*, one can define such a **require** in PURE and because F* expects to be able to lift computations in PURE to any other Dijkstra monad, then such Dijkstra monads should also support a **require** operation.

Partial Dijkstra monads for all. As we pointed out above, computations c in $D\ A\ w$ obtained by DM4All (1) can be coerced to type $M\ A$, by just forgetting the $\theta\ c \leq^{\mathsf{W}} w$ refinement. This means that in order for D to support partiality, the underlying computation monad M should already support partiality. Yet most computation monads do not. For instance, for the state monad, **require** p would need to have type $\mathsf{State}\ p = S \to p \times S$, which one cannot inhabit for an arbitrary $p : \mathbb{P}$.

We show that the DM4All construction can be made to produce partial Dijkstra monads—thus usable in F*—when both the monads M and W additionally support a **require** construct such that $\theta\ (\textbf{require}^{\mathsf{M}}\ p) \leq^{\mathsf{W}} \textbf{require}^{\mathsf{W}}\ p$.

We provide several ways to build computation (and specification) monads that support a **require** construct. First, we provide an account of *Dijkstra monads for free* (DM4Free) [AHM+17] that fits in this setting. Basically, DM4Free produces a partial Dijkstra monad from a computation monad obtained by applying a monad transformer T to the partiality monad $\mathsf{G}\ A = \sum(p : \mathbb{P}).\ (p \to A)$ and the specification monad obtained by applying T to the continuation monad $\mathsf{W}^{\mathsf{Cont}}\ A = (A \to \mathbb{P}) \to \mathbb{P}$. This confirms the empirical observation that DM4Free yields Dijkstra monads that are usable in F*. Second, we provide a construction for adding an extra **require** constructor to the signature of a free monad, allowing for deep occurrences of **require** within computations. Together these cases cover many usual effects such as I/O, non-determinism, state, unrecoverable exceptions, etc. We prove formally in Coq that the DM4All construction with **require** yields partial Dijkstra monads and we include examples of the constructions above.[1] We are also investigating how to adapt interaction trees [XZH+19] to support partiality for potentially non-terminating computations in the style of *Dijkstra monads forever* [SZ21].

[1] https://github.com/TheoWinterhalter/pdm4all/releases/tag/types2022

References

[AHM+17] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 515–529, 2017.

[MAA+19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *PACMPL*, 3(ICFP):104:1–104:29, 2019.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

[SWS+13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

[SZ21] Lucas Silver and Steve Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

[XZH+19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

Pre-bilattices in Univalent Foundations

Georgios V. Pitsiladis*

National Technical University of Athens, Athens, Greece
gpitsiladis@mail.ntua.gr

Abstract

Bilattices are algebraic structures used in logic and artificial intelligence, comprising two lattice orders (usually the one modelling amount of truth and the other modelling amount of information) as well as some property and/or operator that links the two orders. This paper sums up ongoing work on how bilattices can be defined in Univalent Foundations, in particular in the UniMath Coq library.

1 Introduction

1.1 Bilattices

Pre-bilattices are sets equipped with two lattice orderings, usually aimed to model simultaneously the validity of, and degree of knowledge about, sentences from a logical language. They have found applications in diverse fields, among which in multi-valued logics [3, 13], paraconsistent reasoning [2, 3], and logic programming [7, 1], and they have also been studied from an algebraic perspective [5, 10, 6, 9].

In the literature, there are two main ways that the lattices of a pre-bilattice are connected:

- Via a property that includes operations from both orders; the two properties that are usually studied are interlacing and distributivity, which will be described below, but also modularity has been considered [11].
- Via an extra operator, usually negation (a negation on the truth order that is monotonic on the knowledge order) or conflation (a negation on the knowledge order that is monotonic on the truth order). Pre-bilattices with a negation and/or a conflation operator are called bilattices. Also more generic negation-like operators [14] and residuated bilattices [8] have been studied.

Here, we will consider interlacing and distributivity, leaving negation operators (i.e. bilattices) for future work. This is mainly because the most mathematically interesting basic results, the main one being the representation theorem which will be discussed below, are meaningful even at the level of pre-bilattices.

1.2 Univalent Foundations and UniMath

UniMath [16] is a library of formalised mathematics built on the Coq theorem prover [15], using Homotopy Type Theory [12] as its foundation. As such, it's a constructive structural framework in which it is possible to formalise mathematical entities and proofs. Some algebraic structures have already been formalised in UniMath, among them the notion of lattices; in fact, most properties of lattices that will be needed for pre-bilattices are already proven in UniMath.

*Work carried out in the context of PhD supervised by Petros Stefaneas and funded by the Special Account for Research Funding (E.L.K.E.) of National Technical University of Athens.

2 Pre-bilattices in UniMath

The type of lattices in UniMath is a dependent type $\texttt{Lattice} : \mathsf{Set} \to \mathcal{U}$, specifying that a lattice has two associative and commutative binary operators, min ($\sqcap$) and max ($\sqcup$), such that $x \sqcap (x \sqcup y) = x$ and $x \sqcup (x \sqcap y) = x$. Moreover, UniMath defines a partial order ($\leq$) for each lattice, by $x \leq y :\equiv (x \sqcap y = x)$ (a mere proposition, since lattices are defined over sets).

For each $X : \mathsf{Set}$, it is hence possible to define the type of *pre-bilattices*,

$$\texttt{PreBilattice}(\texttt{X}) :\equiv \texttt{Lattice}(\texttt{X}) \times \texttt{Lattice}(\texttt{X}),$$

where the first lattice will be considered the *truth* lattice (with min $\wedge_b$, max $\vee_b$, and ordering $\leq_{\mathrm{t}b}$) and the second lattice will be considered the *knowledge* lattice (with min $\otimes_b$, max $\oplus_b$, and ordering $\leq_{\mathrm{k}b}$).

Notice that there are quite a few dualities in place when studying pre-bilattices: the opposite order of a lattice is again a lattice and, moreover, one can swap the truth and knowledge lattices of a pre-bilattice to obtain another pre-bilattice. As a consequence, there emerges a "prove one, get many" situation, which can be utilised in UniMath to facilitate proving properties of pre-bilattices.

The type of *interlaced* pre-bilattices over a set $\texttt{X}$ is

$$\texttt{InterlacedPreBilattice}(\texttt{X}) :\equiv \sum_{b:\texttt{PreBilattice}(\texttt{X})} \texttt{IsInterlaced}(\texttt{b}),$$

where $\texttt{IsInterlaced}(\texttt{b})$ describes that $\wedge_b$ and $\vee_b$ are monotonic (for their first argument) with respect to $\leq_{\mathrm{k}b}$ and that $\otimes_b$ and $\oplus_b$ are monotonic (for their first argument) with respect to $\leq_{\mathrm{t}b}$.

Similarly, one can define the type of *distributive* pre-bilattices, which are those pre-bilattices such that all pairs of $\wedge_b$, $\vee_b$, $\otimes_b$, and $\oplus_b$ are distributive, i.e. $x \star (y \cdot z) = (x \star y) \cdot (x \star z)$, and prove that distributive pre-bilattices are interlaced.

It is also possible, for each pair $l_1 : \texttt{Lattice}(\texttt{X}_1)$, $l_2 : \texttt{Lattice}(\texttt{X}_2)$ of lattices, to define the product pre-bilattice (for its definition, see for example [4, Definition 3.1]) and prove that it is unique up to equivalence (again, due to the fact that lattices are defined over sets).

The representation theorem. A well-known and important result in the theory of bilattices is the representation theorem, stating that product pre-bilattices are interlaced pre-bilattices and vice-versa. The left-to-right part of the proof follows easily from basic properties. The converse can also be proved in UniMath, formalising the proof in [4, Section 3.1]: it involves defining two equivalence relations on the interlaced pre-bilattice, whose equivalence classes will be the lattices forming the product pre-bilattice. To the author's knowledge, some of the mechanisms in UniMath that are necessary for reasoning with equivalence classes (`setquotunivprop` and the lemmas that depend on it) result in non-computable opaque terms, which implies that the representation theorem will be usable for proofs but not for computations; however, the importance of the theorem lies exactly on the fact that it facilitates proving properties of interlaced pre-bilattices by reducing them to product pre-bilattices.

References

[1] Ofer Arieli. Paraconsistent declarative semantics for extended logic programs. *Annals of Mathematics and Artificial Intelligence*, 36(4):381–417, 2002.

[2] Ofer Arieli. Reasoning with different levels of uncertainty. *Journal of Applied Non-Classical Logics*, 13(3–4):317–343, 2003.

[3] Ofer Arieli and Arnon Avron. Reasoning with logical bilattices. *Journal of Logic, Language and Information*, 5(1):25–63, March 1996.

[4] F. Bou and U. Rivieccio. The logic of distributive bilattices. *Logic Journal of IGPL*, 19(1):183–216, February 2011.

[5] Félix Bou, Ramon Jansana, and Umberto Rivieccio. Varieties of interlaced bilattices. *Algebra universalis*, 66(1–2):115, October 2011.

[6] L. M. Cabrer and H. A. Priestley. Natural dualities through product representations: Bilattices and beyond. *Studia Logica*, 104(3):567–592, 2016.

[7] M. Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programming*, 11(1&2):91–116, 1991.

[8] Ramon Jansana and Umberto Rivieccio. Residuated bilattices. *Soft Computing*, 16(3):493–504, March 2012.

[9] Costas D. Koutras and Georgios V. Pitsiladis. Galois connections for bilattices. *Algebra universalis*, 82(3):37, May 2021.

[10] B. Mobasher, D. Pigozzi, G. Slutzki, and G. Voutsadakis. A duality theory for bilattices. *Algebra universalis*, 43(2–3):109–125, 2000.

[11] Yu M. Movsisyan. Interlaced, modular, distributive and boolean bilattices. *Armenian Journal of Mathematics*, 1(3):7–13, December 2008. Number: 3.

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

[13] U. Rivieccio. Bilattice public announcement logic. In R. Goré, B. P. Kooi, and A. Kurucz, editors, *Advances in Modal Logic 10, invited and contributed papers from the tenth conference on "Advances in Modal Logic," held in Groningen, The Netherlands, August 5–8, 2014*, pages 459–477. College Publications, 2014.

[14] Umberto Rivieccio, Paulo Maia, and Achim Jung. Non-involutive twist-structures. *Logic Journal of the IGPL*, 28(5):973–999, September 2020.

[15] The Coq Development Team. The Coq Proof Assistant, January 2022.

[16] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. available at https://unimath.org .

Proof-relevant normalization for intersection types with profunctors

Zeinab Galal

University of Leeds, UK
z.galal@leeds.ac.uk

Abstract

We present a bicategorical orthogonality construction connecting the idempotent and non-idempotent intersection typing systems for profunctors introduced by Olimpieri. It allows us to translate normalization properties between them by providing explicit reduction 2-cells between the interpretation of terms.

In recent years, there was a growing interest in the categorification of models of computation by replacing semantics where types are sets or preorders with richer categorical structures providing finer mathematical invariants. When using bidimensional categorical structures to model computations, rewriting steps between terms become 2-cells carrying information on reductions between programs. We are interested in recent development on intersection typing systems in this direction: Olimpieri considered the bicategory of profunctors equipped with appropriate monads encoding categorifications of non-idempotent and idempotent intersection typing systems [6, 7]. The bidimensional setting offers a proof-relevant framework as the interpretation of a term contains the set of all its derivations.

We use an orthogonality construction between the two models to reduce normalization for the idempotent case to the non-idempotent one by connecting the interpretation of terms in the two settings with explicit reduction 2-cells. We start by recalling the 1-categorical construction by Ehrhard which we generalize to the bicategorical setting. In the 1-categorical case, there is a combinatorial proof for head normalization by extracting an upper bound on the number of head reductions to its normal form [1]. This construction can be extended to the bicategorical setting where Olimpieri further exhibits a reduction isomorphism between the interpretation of a term and its head normal form for profunctors with the non-idempotent monad [6, 7]. On the other hand, the proof uses Tait's reducibility argument for the idempotent case and we lose the reduction witness. One of the applications of our construction is that we now obtain a witness 2-cell for the idempotent setting as well whereas in the 1-dimensional setting, the orthogonality constuction gives no information on the reduction beyond existence.

In order to reduce the proof of normalization for the idempotent case to the combinatorial one for the 1-categorical case, Ehrhard uses an orthogonality connecting the two models of linear logic **Rel** (the category of sets and relations) and **ScottL** (the category of preorders and ideal relations) [3, 2]. For a preorder $P = (|P|, \leq_P)$ and two subsets $x, x' \subseteq |P|$, the orthogonality is defined as follows

$$x \perp_P x' \quad :\Leftrightarrow \quad (x \cap x' \neq \emptyset \Leftrightarrow \downarrow(x) \cap \uparrow(x') \neq \emptyset).$$

It induces a Galois connection: for a subset $X \subseteq \mathcal{P}(|P|)$, its orthogonal $X^\perp$ is given by the set $\{x' \subseteq |P| \mid \forall x \in X, x \perp_P x'\}$. Ehrhard defines a new category **Pop** (preorders with projections) whose objects are pairs (P, X) of a preorder P and a subset $X \subseteq \mathcal{P}(|P|)$ such that $X = X^{\perp\perp}$. Intuitively, X contains all the denotations x of terms in **Rel** that can be mapped to **ScottL** by taking their downclosure $\downarrow(x)$. This new category allows to prove that the interpretation of a term is not empty in **Rel** if and only if it is not empty in **ScottL** and this equivalence is

crucially used to translate the combinatorial normalization theorem from the non-idempotent intersection typing system to the idempotent one [2].

In the categorified setting, we replace sets by groupoids and preorders by categories so that the operation mapping a preorder to its underlying set $P \mapsto |P|$ now corresponds to taking the *core* of category (for a small category $\mathbb{A}$, its *core*, denoted by $\mathbf{c}\,\mathbb{A}$, is the maximal subgroupoid of $\mathbb{A}$). Relations $R \subseteq A \times B$ (or equivalently functions $A \times B \to \{0,1\}$) become profunctors $P : \mathbb{A} \nrightarrow \mathbb{B}$ (or equivalently functors $\mathbb{A} \times \mathbb{B}^{\mathrm{op}} \to \mathbf{Set}$). To represent intersection types, we use finite sequences instead of finite multisets with the free symmetric strict monoidal completion $\mathcal{S}\mathbb{A}$ and the free coproduct completion $\mathcal{C}\mathbb{A}$ for a category $\mathbb{A}$.

Definition 0.1. For a small category $\mathbb{A}$, define $\mathcal{C}\mathbb{A}$ to be the category whose objects are finite sequences $\langle a_1, \ldots, a_n\rangle$ of objects of $\mathbb{A}$ and a morphism between two sequences $\langle a_1, \ldots, a_n\rangle$ and $\langle b_1, \ldots, b_m\rangle$ consists of a pair $(\sigma, (f_i)_{i \in \underline{n}})$ of a function $\sigma : \underline{n} \to \underline{m}$ and a family of morphisms $f_i : a_i \to b_{\sigma(i)}$ in $\mathbb{A}$ for $i \in \underline{n}$. The category $\mathcal{S}\mathbb{A}$ has the same objects but morphisms $(\sigma, (f_i)_{i \in \underline{n}})$ are restricted to the ones where σ is a bijection.

In $\mathcal{S}\mathbb{A}$ there are no morphisms between sequences of different lengths which encodes non-idempotency whereas $\mathcal{C}\mathbb{A}$ allows for duplication and erasure and is used for the idempotent case. In our setting, we consider two bicategories $\mathbf{ProfG}_{\mathcal{S}}$ and $\mathbf{Prof}_{\mathcal{C}}$ representing the categorifications of **Rel** and **ScottL** respectively [4, 5]. The objects of $\mathbf{ProfG}_{\mathcal{S}}$ are small groupoids, the morphisms are profunctors of the form $\mathcal{S}\mathbb{A} \nrightarrow \mathbb{B}$ and the 2-cells are natural transformations between them whereas the objects of $\mathbf{Prof}_{\mathcal{C}}$ are small categories, the morphisms are profunctors of the form $\mathcal{C}\mathbb{A} \nrightarrow \mathbb{B}$ and the 2-cells are natural transformations. Olimpieri studied the idempotent and non-idempotent intersection typing systems associated to the bicategories $\mathbf{Prof}_{\mathcal{C}}$ and $\mathbf{ProfG}_{\mathcal{S}}$ respectively [6, 7].

Similarly to the 1-categorical case, we connect the interpretation of terms in these two bicategories using an orthogonality construction. For a category $\mathbb{A}$ and profunctors $X : \mathbf{1} \nrightarrow \mathbf{c}\,\mathbb{A}$, $X' : \mathbf{c}\,\mathbb{A} \nrightarrow \mathbf{1}$, the orthogonality carries a 2-cell retraction $r : X' \downarrow_{\mathbb{A}} X \Rightarrow X'X$ which intuitively provides a witness connecting the idempotent setting and the non-idempotent setting. The Galois connection in the 1-categorical case now becomes a contravariant adjunction between slice categories and the objects of the new bicategory that we consider are the fixpoints of this adjunction (corresponding to closure under double-orthogonality). We show that we obtain a cartesian closed bicategory and solve fixpoint equations which allows us to obtain explicit retraction 2-cells connecting the interpretations in the idempotent and non-idempotent cases as desired. Since our construction provides us with explicit reduction 2-cells for the idempotent setting, we aim to study execution time by translating existing results in the non-idempotent case [1].

Another advantage of this construction already highlighted by Ehrhard in the 1-categorical setting is its modularity. We can encode various calculi (PCF, standard λ-calculus, bang calculus, call-by-value λ-calculus, etc.) by considering solutions of different fixed point equations. For example, Ehrhard considers a call-by-value calculus by solving the equation $D = !D \multimap !D$ whereas Olimpieri considers retractions $D \lhd !D \multimap D$. If we want to prove normalization for the induced idempotent intersection typing systems, we need to use reducibility candidates for each of the fixpoint equations we consider. The orthogonality construction provides a more modular framework as we know that the result holds for any fixpoint equation obtained using linear logic connectives. In the proof-relevant bicategorical setting, we further automatically have the reduction 2-cell connecting the idempotent and non-idempotent interpretations for all fixpoint equations. Our end goal is also to develop a theory of orthogonality for bidimensional structures enabling us to control interactions between terms and environments and also to restrict the allowed reductions between terms.

References

[1] Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. PhD dissertation, Université Aix-Marseille 2, 2008.

[2] Thomas Ehrhard. Collapsing non-idempotent intersection types. *Leibniz International Proceedings in Informatics, LIPIcs*, 16, 09 2012.

[3] Thomas Ehrhard. The Scott model of Linear Logic is the extensional collapse of its relational model. *Theoretical Computer Science*, 424:20–45, 2012. 26 pages.

[4] Marcelo Fiore, Nicola Gambino, Martin Hyland, and Glynn Winskel. The cartesian closed bicategory of generalised species of structures. *J. Lond. Math. Soc. (2)*, 77(1):203–220, 2008.

[5] Zeinab Galal. A profunctorial Scott semantics. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[6] Federico Olimpieri. *Intersection types and resource calculi in the denotational semantics of lambda-calculus*. PhD thesis, Aix-Marseille, 2020.

[7] Federico Olimpieri. Intersection type distributors. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–15. IEEE, 2021.

Quantitative Inhabitation in Call-by-Value

Victor Arrial

Université Paris Cité - IRIF - CNRS

Abstract

We show the decidability of the inhabitation problem in a quantitative Call-by-Value setting.

Inhabitation. *Type systems* are formalisms made of rules assigning a *type* to the constructs of a programming language, usually represented by a *term calculus*. Types enforce some particular specification (*e.g.* termination, memory safety, deadlock freeness, etc), so that they guarantee the construction of well-behaved terms, in the sense that "well-typed programs cannot go wrong" [15]. Typing in a given type system $\mathcal{X}$ is written $\triangleright_{\mathcal{X}} \Gamma \vdash t : \sigma$, where t is a term, σ is the type assigned to t, and Γ is an *environment* assigning types to the (free) variables of t. The *inhabitation* problem naturally arises for any given type assignment system: given an environment Γ, and a type σ, decide whether there exists a term such that $\triangleright_{\mathcal{X}} \Gamma \vdash t : \sigma$. Inhabitation corresponds to decide the existence of a program (term t) that satisfies the given specification (type σ) under additional assumptions (environment Γ). Decidability of the inhabitation problem naturally provides tools for type-based *program synthesis* [14, 3], whose task is to construct—from scratch—a program that satisfies some high-level formal specification (the one guaranteed by the type assignment).

Quantitative Typing Systems. *Intersection type assignment systems* [8, 9] were introduced for the λ-calculus to increase the typability power of simple types by introducing a new *intersection* type constructor $\wedge$, which is, in principle, associative, commutative and *idempotent* ($\sigma \wedge \sigma = \sigma$). Intersection types allow terms having different types simultaneously, *e.g.* a term t has type $\sigma \wedge \tau$ whenever t has both the type σ and the type τ. In these (idempotent) systems typability and inhabitation are both undecidable [17]. However, intersection types constitute a powerful tool to reason about *qualitative* properties of programs, for example, there are intersection type systems characterizing different notions of normalization [16, 10], in the sense that a term t is typable in a given system if and only if t is normalizing for some particular notion. By removing idempotence [13, 11], a term of type $\sigma \wedge \sigma \wedge \tau$ can be seen as a resource that, during execution, can be used once as a data of type τ and twice as a data of type σ. The resulting *non-idempotent* type systems do not only provide qualitative characterization of operational properties, but also *quantitative* ones, in the sense that a term t is still typable if and only if is normalizing, and in addition, any type derivation of t gives a *bound* to the execution time for t (the number of steps to reach a normal form) [12, 1]. In such a setting, typability is still undecidable, nevertheless inhabitation has proven to be *decidable* in the Call-by-Name case [6, 7]. So, an algorithm solving the inhabitation problem for a quantitative type system provides a decidable tool for type-based *quantitative program synthesis*, which aims to construct—from scratch—a program that satisfies some quantitative specification.

Call-by-Value. Call-by-Value evaluation is the most common parameter passing mechanism for programming languages: parameters are evaluated before being passed. We use the $\lambda_{\texttt{vsub}}$ calculus introduced by Accattoli and Paolini [2] which exploits explicit substitutions for both delaying CBV redex restrictions as well as acting at a distance, and which thus presents good

operational properties. This calculus can be equipped with a non-idempotent intersection type system [4] (presented in Fig. 1) which characterizes head normalization. We use multisets to denote intersections.

$$\frac{}{x : \mathcal{M} \vdash x : \mathcal{M}}\ \mathsf{ax} \qquad \frac{(\ \Gamma_i; x : \mathcal{M}_i \vdash t : \sigma_i)_{i\in I}}{+_{i\in I}\Gamma_i \vdash \lambda x.t : [\mathcal{M}_i \Rightarrow \sigma_i]_{i\in I}}\ \mathsf{abs}$$

$$\frac{\Gamma_1 \vdash t : [\mathcal{M} \Rightarrow \sigma] \quad \Gamma_2 \vdash u : \mathcal{M}}{\Gamma_1 + \Gamma_2 \vdash tu : \sigma}\ \mathsf{app} \qquad \frac{\Gamma_1; x : \mathcal{M} \vdash t : \sigma \quad \Gamma_2 \vdash u : \mathcal{M}}{\Gamma_1 + \Gamma_2 \vdash t[x\backslash u] : \sigma}\ \mathsf{es}$$

Figure 1: Call-by-Value Type System $\mathcal{V}$

Contributions. We show that the inhabitation problem for the Call-by-Value λ-calculus with respect to the quantitative type system $\mathcal{V}$ [4] is decidable. We do not simply give an algorithm searching for a term that can be typed with a given environment Γ and type σ, but we solve a more ambitious goal: we look for *all* and *only* such typable terms. This provides either a strong and powerful tool for (quantitative) program synthesis.

Canonical Derivations as Finite Basis: The set of all solutions of any instance of the inhabitation problem is either infinite or empty. Building an algorithm providing all solutions for a given instance is therefore worthless. However, following the technique used in the Call-by-Name λ-calculus case [6, 7] and generalizing it to explicit substitutions [5] allows us to introduce the notion of *canonical derivation* which later forms a finite basis for the solution set of each instance. It is a basis as it *exactly generates* each solution set through two simple operations: redex expansion and term plugging. We show how to compute the canonical representative of any solution as well as how it is recovered from its canonical representative, thus providing proofs of *correction* and *completeness* for each basis.

Basis Search Algorithm: Equipped with such tools, we provide an algorithm computing the basis for any given instance. It highly exploits a central property of any basis of terms called *head subtype property* which indicates that some of its types are contained in the context and thus helps guiding the search. This (non-deterministic) algorithm is shown to be *correct* and *complete* as it finds all and only basis terms. We show that it can in fact be *deterministically simulated* in a finite time, which provides for free a proof of the finiteness of each basis. It therefore constitutes a program synthesis mechanism for an expressive programming language.

References

[1] Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds, fully developed. J. Funct. Program., 30:e14, 2020.

[2] Beniamino Accattoli and Luca Paolini. Call-by-value solvability, revisited. In Tom Schrijvers and Peter Thiemann, editors, FLOPS, volume 7294 of LNCS, pages 4–16. Springer, 2012.

[3] Jan Bessai, Tzu-Chun Chen, Andrej Dudenhefner, Boris Düdder, Ugo de'Liguoro, and Jakob Rehof. Mixin composition synthesis based on intersection types. CoRR, abs/1712.06906, 2017.

[4] Antonio Bucciarelli, Delia Kesner, Alejandro Ríos, and Andrés Viso. The bang calculus revisited. In Keisuke Nakano and Konstantinos Sagonas, editors, Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings, volume 12073 of Lecture Notes in Computer Science, pages 13–32. Springer, 2020.

[5] Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Solvability = Typability + Inhabitation. Logical Methods in Computer Science, Volume 17, Issue 1, January 2021.

[6] Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. The inhabitation problem for non-idempotent intersection types. In TCS, volume 8705 of LNCS, pages 341–354. Springer, 2014.

[7] Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Inhabitation for non-idempotent intersection types. Logical Methods in Computer Science, 14(3), 2018.

[8] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. Archiv für mathematische Logik und Grundlagenforschung, 19(1):139–156, 1978.

[9] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. Notre Dame J. Form. Log., 21(4):685–693, 1980.

[10] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. Math. Log. Q., 27(2-6):45–58, 1981.

[11] Daniel de Carvalho. Sémantiques de la logique linéaire et temps de calcul. PhD thesis, Université Aix-Marseille II, 2007.

[12] Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. Math. Struct. Comput. Sci., 28(7):1169–1203, 2018.

[13] Philippa Gardner. Discovering needed reductions using type theory. In Masami Hagiya and John C. Mitchell, editors, Theoretical Aspects of Computer Software, pages 555–574, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[14] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst., 2(1):90–121, 1980.

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

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

[17] Pawel Urzyczyn. The emptiness problem for intersection types. Journal of Symbolic Logic, 64(3):1195–1215, 1999.

Quantitative Perspectives on Generalized Applications

Loïc Peyrot

Université de Paris, CNRS, IRIF, Paris, France
lpeyrot@irif.fr

Joachimski and Matthes λ-*calculus with generalized applications* ΛJ [10, 11] is a Curry-Howard interpretation of von Plato's natural deduction with generalized elimination rules [16]. The difference between ΛJ and the usual λ-calculus lies in the application constructor: its structure is generalized to $t(u, y.r)$, where t, u, r are subterms and y is a variable bound in r. This extended form comes from the generalized implication elimination rule:

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A \qquad \Gamma, y : B \vdash r : C}{\Gamma \vdash t(u, y.r) : C}\ (\to_e)$$

Intuitively, $y.r$ can be seen as a "continuation" waiting for the result of the application of t to u. Once this result is available, it will be substituted in r for y. This can be seen in the following β_j-rule:

$$(\lambda x.t)(u, y.r) \to_{\beta_j} \{\{u/x\}t/y\}r,$$

where the result is simply the β-reduction of $(\lambda x.t)u$. An additional permutative rule π: $t(u, x.r)(u', y.r') \to_\pi t(u, x.r(u', y.r'))$ enables to reduce terms to a *fully normal form*, where the left side of an argument is always a variable, such as in the term $x(u, y.r)$. The calculus ΛJ has its origins in natural deduction, but it also has links with the sequent calculus [8] and calculi with explicit substitutions [12].

In this talk, I look at generalized applications through the lens of *quantitative types*. Quantitative types, also known as non-idempotent intersection types, were introduced by Gardner [9], and later independently by Kfoury [14] and De Carvalho [4]. Like in Coppo and Dezani's idempotent intersection types [3], normalization of some reduction relation $\mathcal{R}$ can be characterized by typability in a quantitative system: a term normalizes for $\mathcal{R}$ *iff* it is typable in an appropriate type system. This semantical flavor is of great use to easily derive equivalences between different operational relations. On top of these qualitative properties, non-idempotence brings a quantitative analysis. In particular, the size of type derivations usually provides an upper bound on the length of the longest reduction sequence to normal form.

An *idempotent* intersection type system for ΛJ was already defined by Matthes [15]. But the change of perspective to non-idempotence gives new insights on the calculus and leads us to define a variant λJ_n[1], differing from the original calculus in two main ways.

First, we use a *distant* β_j ($\mathtt{d}\beta_j$) rule, where distance is a paradigm we adopt from calculi with explicit substitutions [1]. As in these calculi, some redexes of ΛJ can be hidden by the syntax, such as in the term $x(u, y.\lambda z.r)(u', y'.r')$, where the computation between $\lambda z.r$ and u' is stuck. Conversion π enables to unblock this redex:

$$x(u, y.\lambda z.r)(u', y'.r') \to_\pi x(u, y.(\lambda z.r)(u', y'.r')) \to_{\beta_j} x(u, y.\{\{u'/z\}r/y'\}r').$$

But in our approach, permutations are simply considered as a tool to overcome syntactical limitations, leaving the computational content inside the β_j-reduction itself. Indeed, only β_j-reductions create divergence. Such an approach is motivated by graphical formalisms such as

[1] The subscript n emphasizes the call-by-name nature of the calculus, as a promising call-by-value calculus with generalized applications was recently defined by Espírito Santo [5].

proof nets, where these kinds of syntactical obstructions do not occur. Concretely, we only integrate the necessary permutations in the distant β_j rule, so that the calculus relies on a unique computational step. This is more relevant for the quantitative analysis, since typing should not be affected by permutations.

The second divergence between our approach and ΛJ is that distance is not in fact based on the usual permutation π. This rule is not sound for a quantitative semantics: subject reduction fails in our type system. Intuitively, π introduces a sharing of subterms fit for a call-by-value calculus, rather than call-by-name like ΛJ. While in the call-by-name λ-calculus it is enough to reject permutative rules altogether, the situation is a bit more complex with generalized applications: we still need to unblock redexes. We choose to base the distant rule on a different permutation called p_2: $t(u, y.\lambda x.r) \to_{p_2} \lambda x.t(u, y.r)$, originating from previous studies on generalized applications [6]. Our previous example is then reduced in one step as: $x(u, y.\lambda z.r)(u', y'.r') \to_{\mathtt{d}\beta_j} \{x(u, y.\{u'/z\}r)/y'\}r'$.

In summary, we replace the two original rules β_j and π by a single computational one based on β_j and permutation p_2. We show that the new calculus λJ_n is sound and complete with respect to the quantitative semantics by characterizing strong normalization with a quantitative type system. We also show the equivalence of strong normalization for ΛJ and λJ_n, by introducing inductive definitions of strong normalization. These results appear in [7], in collaboration with José Espírito Santo and Delia Kesner.

After helping in revisiting the dynamics of the general calculus, quantitative types are still useful to analyze finer reduction relations. A notable one is *head* reduction, which characterizes *solvability* operationally: a term is solvable *iff* it has a head-normal form [17]. Solvability is a crucial property identifying *meaningful* terms [2], *i.e.* terms that can influence the observational behavior of a reduction. Denotationally, a model of the λ-calculus identifying all unsolvable terms (*e.g.* as bottom) is consistent, while identifying only non-normalizing terms is not. In [13], Delia Kesner and I give a reduction relation, generalizing head reduction, which characterizes an original notion of solvability for generalized applications operationally. We then define an appropriate quantitative type system. This logical characterization enables us to prove preservation of solvability to and from the λ-calculus, by a simple proof of preservation of typability.

References

[1] Beniamino Accattoli and Delia Kesner. The structural λ-calculus. In *Computer Science Logic*, pages 381–395. Springer Berlin Heidelberg, 2010.

[2] Henk Pieter Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. Elsevier, 1984.

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

[4] Daniel De Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, January 2017.

[5] José Espírito Santo. The call-by-value lambda-calculus with generalized applications. In *28th EASCL Annual Conference on Computer Science Logic (CSL 2020)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2020.

[6] José Espírito Santo and Luís Pinto. Permutative conversions in intuitionistic multiary sequent calculi with cuts. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, pages 286–300. Springer, 2003.

[7] José Espírito Santo, Delia Kesner, and Loïc Peyrot. A faithful and quantitative notion of distant reduction for generalized applications. In *Proc. of the 24th Int. Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, LNCS, 04 2022.

[8] José Espírito Santo and Luís Pinto. A calculus of multiary sequent terms. *ACM Transactions on Computational Logic*, 12(3):1–41, May 2011.

[9] Philippa Gardner. Discovering needed reductions using type theory. In Masami Hagiya and John C. Mitchell, editors, *Lecture Notes in Computer Science*, pages 555–574. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[10] Felix Joachimski and Ralph Matthes. Standardization and confluence for a lambda calculus with generalized applications. In *Rewriting Techniques and Applications*, pages 141–155. Springer Berlin Heidelberg, 2000.

[11] Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic*, 42(1):59–87, 2003.

[12] Delia Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3), 2009.

[13] Delia Kesner and Loïc Peyrot. Solvability for generalized applications. Unpublished, 2022.

[14] Assaf Kfoury. A linearization of the lambda-calculus and consequences. *Journal of Logic and Computation*, 10(3):411–436, 2000.

[15] Ralph Matthes. Characterizing strongly normalizing terms for a lambda calculus with generalized applications via intersection types. In *Proc. ICALP 2000 (Geneva), volume 8 of Proceedings in Informatics*, pages 339–353, 2000.

[16] Jan von Plato. Natural deduction with general elimination rules. *Archive for Mathematical Logic*, 40(7):541–567, 2001.

[17] Christopher P. Wadsworth. The relation between computational and denotational properties for scott's D_∞-models of the lambda-calculus. *SIAM Journal on Computing*, 5(3):488–521, September 1976.

Realizing Implicit Computational Complexity*

Clément Aubert[1], Thomas Rubiano[2], Neea Rusch[1], and Thomas Seiller[2,3]

[1] School of Computer and Cyber Sciences, Augusta University, USA
[2] LIPN – UMR 7030 Université Sorbonne Paris Nord, France
[3] CNRS, France

Originalities in Implicit Computational Complexity. Automatic performance analysis and optimization is a critical for systems with resource constraints. The field of Implicit Computational Complexity (ICC) [12] pioneers in embedding in the program itself a guarantee of its resource usage, using e.g. bounded recursion [10, 18] or type systems [8, 17]. It drives better understanding of complexity classes, but also introduces original methods to develop resources-aware languages, static source code analyzers and optimizations techniques, often relying on informative and subtle type systems. Among the methods developed, the *mwp-flow analysis* [16] certifies polynomial bounds on the size of the values manipulated by an imperative program, obtained by bounding the transitions between states instead of focusing on states in isolation, and is not concerned with termination or tight bounds on values. It introduces a new way of tracking dependencies between "chunks" of code by typing each statement with a matrix listing the way variables relate to each others.

Having introduced such novel analysis techniques, and, as opposed to traditional complexity, by utilizing models that are generally expressive enough to write down actual algorithms [20, p. 11], ICC provides a conceivable pathway to automatable complexity analysis and optimization. However, the approaches have rarely materialized into concrete programming languages or program analyzers extending beyond toy languages, with a few exceptions [7, 15]. Absence of realized results reduces ability to test the true power of these techniques, limits their application in general, and understanding their capabilities and potential expressivity remains underexplored.

We present an ongoing effort to address this deficiency by applying the mwp-flow analysis, that tracks dependencies between variables, in three different ways, at different stages of maturation, in their temporal order. The first and third projects bend this typing discipline to gain finer-grained view on statements independence, to optimize loops by hoisting invariant [21] and by splitting loops "horizontally" to parallelize them [5]. The second project refines, extends and implements the original analysis to obtain a fast, modular static analyzer [6]. All three projects aim at pushing the original type system to its limits, to assess how ICC can in practice lead to original, sometimes orthogonal, approaches. We also discuss our intent and motivations behind formalizing this analysis using Coq proof assistant [22], in a spearheading endeavour toward formalizing complexity analysis.

1. Loop Quasi-Invariant Chunk Detection. Loop peeling for hoisting (quasi-)invariants can be used to optimize generated code [1, p. 641], and is implemented e.g. in LLVM as the `licm` pass. This work [21] leverages an ICC-inspired dependency analysis to provide a transformation method to compilers. It enables detection of quasi-invariants of arbitrary degree in composed statements called "chunks". It reuses the mwp's matricial system and typed data flows to generate dependency graphs, to compute an invariance degree for each statement or chunks of

*This research is supported by the Th. Jefferson Fund of the Embassy of France in the United States and the FACE Foundation, and has benefited from the research meeting 21453 "Static Analyses of Program Flows: Types and Certificate for Complexity" in Schloss Dagstuhl. Th. Rubiano and Th. Seiller are supported by the Île-de-France region through the DIM RFSI project "CoHOp".

statements. It then finds the maximum (worst) dependency graph for loops, and recognizes whether an entire block is quasi-invariant. If this block is an inner loop, it can be hoisted, and the computational complexity of the overall program can be decreased. A prototype analysis pass [3] has been designed, proven correct and successfully implemented using a toy `C` parser, and as a prototype pass for the LLVM. This is the first known application of introducing ICC techniques in mainstream compilers.

2. Improved and Implemented mwp-Analysis. In an ongoing development, we improved and implemented the mwp-bounds analysis [16], which certifies that the values computed by an imperative program are bounded by polynomials in the program's input, represented in a matrix of typed flows, characterizing controls from one variable to another. While this flow analysis is elegant and sound, it is also computationally costly–it manipulates non-deterministically a potentially exponential number of matrices in the size of the program [6, 2.3]—and missed an opportunity to leverage its built-in compositionality. We addressed both issues by expanding the original flow calculus, and adjusting its internal machinery to enable tractable analysis [6], and further extended the theory with analysis of function definitions and calls—including recursive ones, a feature not widely supported [14, p. 359]. Our effort and theoretical development is realized in an open-source tool `pymwp` [4], capable of automatically analyzing complexity of programs written in a subset of the `C` programming language.

3. Splitting Loops Horizontally to Improve Their Parallel Treatment. Our most recent effort is directed toward program optimization through loop parallelization. Using an ICC-inspired data flow-based variable dependency analysis, we can reproduce the *tour de force* of detecting opportunities for loop fission that have been missed by other standard analyses [21]. In particular, the dependency analysis allows optimizing loops by splitting them "horizontally", e.g. from `for (int i = 1; i < 10; i++){a[i] = a[i-1] + i; b[i] = b[i-1] + i;}` to:

```
for (int i = 1; i < 10; i++){a[i] = a[i-1] + i;}
for (int i = 1; i < 10; i++){b[i] = b[i-1] + i;}
```

Our approach can process loop-carried dependencies [11, 3.5.2]–such as the one illustrated above–and optimizes `while` loops [5, Sect. 5].—and, more generally, loops whose trip-count cannot be known at compilation time—that are completely ignored by OpenMP [11, 3.2.2], and generally present great difficulty and often prevent optimization [5, Sect. A]. Combined with OpenMP `pragma` directives, this approach gives a speed-up "as good as" `AutoPar-Clava`—which "compare[s] favorably with closely related auto-parallelization compilers" [2, p. 1]—when both are applicable that can be integrated in automatic parallelization pipelines [5, Sect. B]. Our benchmark, shared at `https://github.com/statycc/icc-fission`, substantiate experimentally those claims and provides further evidence.

. . . and Pushing Even Further. From there, many other directions can be explored. Since ICC techniques tend to be designed for simpler program syntax, compiler intermediate representations present an ideal location and point of integration for performing analyses. Implementing the analysis in certified tools such as the CompCert compiler [19] (or, more precisely, its static single assignment version [9]) naturally necessitates certifying the complexity analysis, and we plan to pursue this effort using the Coq proof assistant [22]. The plasticity of both compilers and of the implemented analysis should facilitate porting our results to support further programming languages in addition to `C`. As complexity analysis is difficult in Coq [13], we believe a push would be welcome, and that ICC provides the necessary tools for it.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Hamid Arabnejad, João Bispo, João M. P. Cardoso, and Jorge G. Barbosa. Source-to-source compilation targeting openmp-based automatic parallelization of C applications. *J. Supercomput.*, 76(9):6753–6785, Sep 2020. doi:10.1007/s11227-019-03109-9.

[3] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. Lqicm on c toy parser. URL: https://github.com/statycc/LQICM_On_C_Toy_Parser.

[4] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. pymwp: MWP analysis in Python. URL: https://github.com/statycc/pymwp/.

[5] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. A Novel Loop Fission Technique Inspired by Implicit Computational Complexity. Submitted to ATVA 2022, May 2022. URL: https://hal.archives-ouvertes.fr/hal-03669387.

[6] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. mwp-analysis improvement and implementation: Realizing implicit computational complexity. In Amy Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD)*, LIPIcs. Schloss Dagstuhl, March 2022. To appear. URL: https://hal.archives-ouvertes.fr/hal-03596285.

[7] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP):43:1–43:29, 2017. doi:10.1145/3110287.

[8] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE Computer Society, 2004. doi:10.1109/LICS.2004.1319621.

[9] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, 2014. doi:10.1145/2579080.

[10] Stephen J. Bellantoni and Stephen Arthur Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *STOC*, pages 283–93. ACM, 1992. doi:10.1145/129712.129740.

[11] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, Oxford, England, October 2000.

[12] Ugo Dal Lago. A short introduction to implicit computational complexity. In Nick Bezhanishvili and Valentin Goranko, editors, *ESSLLI*, volume 7388 of *LNCS*, pages 89–109. Springer, 2011. doi:10.1007/978-3-642-31485-8_3.

[13] Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes)*. PhD thesis, Inria, Paris, France, 2019. URL: https://tel.archives-ouvertes.fr/tel-02437532.

[14] Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen. Complexityparser: An automatic tool for certifying poly-time complexity of Java programs. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *LNCS*, pages 357–365. Springer, 2021. doi:10.1007/978-3-030-85315-0_20.

[15] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, pages 781–786. Springer, 2012. doi:10.1007/978-3-642-31424-7_64.

[16] Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4):28:1–28:41, 2009. doi:10.1145/1555746.1555752.

[17] Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1):163–180, 2004. doi:10.1016/j.tcs.2003.10.018.

[18] Daniel Leivant. Stratified functional programs and computational complexity. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, pages 325–333. ACM Press, 1993. doi:10.1145/158511.158659.

[19] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

[20] Jean-Yves Moyen. *Implicit Complexity in Theory and Practice*. Habilitation thesis, University of Copenhagen, 2017. URL: https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf.

[21] Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk detection. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *LNCS*. Springer, 2017. doi:10.1007/978-3-319-68167-2_7.

[22] The Coq Development Team. The coq proof assistant, version 8.7.0, October 2017. doi:10.5281/zenodo.1028037.

Resizing Prop down to an axiom

Stefan Monnier[1]

Université de Montréal - DIRO, Montréal, Canada
monnier@iro.umontreal.ca

Abstract

We present an axiomatization of the impredicative Prop universe of the Calculus of Constructions. More specifically, we present two extensions of a predicative CCω, one with an impredicative quantification in the bottom universe, and the other with a set of axioms and related reduction rules, and then we prove equivalence of the two calculi.

As the title suggests, the axioms are closely related to HoTT's propositional resizing axiom, thus giving concrete evidence for the intuition that this form of impredicativity is equivalent to that offered by the Prop universe. This can also help interoperability between proof systems with a Prop universe and those without.

We finish by discussing how this result can be extended to a calculus with inductive types.

1 Introduction

Impredicativity, in the context of type theory, is the ability for a proposition to be quantified over a type which can be instantiated with that very same proposition. The kind of circularity it introduces is more subtle than that of recursion, but just as dangerous. While many systems disallow this principle outright, favoring the simpler and cleaner metatheory of predicative type theories, it is quite popular both in the context of proof assistants and in the context of functional programming languages.

There are a few different ways to introduce impredicativity, such as: via $\mathsf{Type} : \mathsf{Type}$, as used in Dependent Haskell [7]; via the traditional impredicative bottom universe, as used in System F and in the Calculus of Constructions [3]; via resizing axioms, as used in the HoTT book [6]; or via universe polymorphism, as proposed in [4].

Of those, the impredicative bottom universe as implemented in Coq's Prop is the most widely used in proof assistants and it is generally accepted that the propositional resizing axiom is an alternative to it of comparable power. Yet there is as yet no formal investigation to show precisely *how* comparable they are.

We intend to remedy this situation in the present work in the following way: we show a proof of equivalence between two calculi we call $\mathsf{iCC}\omega$ and $\mathsf{rCC}\omega$: both are pure type system (PTS) [1] with the usual tower of universes; the first comes with an impredicative quantification in universe Type_0 while the second replaces it with a set of axioms which provides a form of propositional resizing. More specifically our contributions are:

- A type-preserving translation of any term of $\mathsf{iCC}\omega$ into a term of $\mathsf{rCC}\omega$, following the approach of syntactic models used in [2].
- A type-preserving translation of any term of $\mathsf{rCC}\omega$ into a term of $\mathsf{iCC}\omega$, simply by providing definitions for the axioms.
- An extension of these results to calculi with inductive types. This extension is not fully satisfactory because the equivalence is not fully satisfied.

2 The encoding

The calculi we use in this paper are pure type systems [1], using the following syntax:

$$\begin{array}{llll} \textit{(var)} & x, y, t & \in & \mathcal{V} \\ \textit{(sort)} & s & \in & \mathcal{S} \\ \textit{(term)} & e, \tau & ::= & s \mid x \mid (x : \tau_1) \to \tau_2 \mid \lambda x : \tau . e \mid e_1 @ e_2 \end{array}$$

A specific PTS is then defined by providing the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ which defines respectively the sorts, axioms, and rules of this system. All our calculi are extensions of the following base predicative calculus we call $\mathsf{pCC}\omega$:

$$\begin{array}{lcll} \mathcal{S} & = & \{\ \mathsf{Type}_\ell & \mid\ \ell \in \mathbb{N}\ \} \\ \mathcal{A} & = & \{\ (\mathsf{Type}_\ell : \mathsf{Type}_{\ell+1}) & \mid\ \ell \in \mathbb{N}\ \} \\ \mathcal{R} & = & \{\ (\mathsf{Type}_{\ell_1}, \mathsf{Type}_{\ell_2}, \mathsf{Type}_{\mathsf{max}(\ell_1,\ell_2)}) & \mid\ \ell_1, \ell_2 \in \mathbb{N}\ \} \end{array}$$

$\mathsf{iCC}\omega$ is defined as the extension of $\mathsf{pCC}\omega$ with the following rules:

$$\mathcal{R} \quad = \quad \ldots \cup \{\ (\mathsf{Type}_\ell, \mathsf{Type}_0, \mathsf{Type}_0)\ \mid\ \ell \in \mathbb{N} \wedge \ell > 0\ \}$$

As a first approximation, $\mathsf{rCC}\omega$ is defined as the extension of $\mathsf{pCC}\omega$ with the following axioms and rewrite rules:

$$\begin{array}{lcll} ||\cdot|| & : & \mathsf{Type}_\ell \to \mathsf{Type}_0 & \text{for all } \ell \in \mathbb{N} \\ |\cdot| & : & (t : \mathsf{Type}_\ell) \to t \to ||t|| & \text{for all } \ell \in \mathbb{N} \\ \mathsf{bind} & : & (t_1 : \mathsf{Type}_{\ell_1}) \to (t_2 : \mathsf{Type}_{\ell_2}) \to ||t_1|| \to (t_1 \to ||t_2||) \to ||t_2|| & \text{for all } \ell_1, \ell_2 \in \mathbb{N} \end{array}$$

$$\mathsf{bind}\ |e_1|\ \lambda x : \tau . e_2 \quad \rightsquigarrow \quad e_2\{e_1/x\}$$

Where $||\cdot||$ is pronounced "erased" and $|\cdot|$ is pronounced "erase". The crux of the matter to encode (written $[\cdot]$) an impredicative quantification $(x : \tau_1) \to \tau_2$ of $\mathsf{iCC}\omega$ into $\mathsf{rCC}\omega$ is to erase it with $||\cdot||$ so as to bring it back down to Type_0: $[(x : \tau_1) \to \tau_2] = ||(x : [\tau_1]) \to [\tau_2]||$
As a consequence, the encoding needs to erase systematically all inhabitants of the Type_0 universe. A first cut of the encoding looks like the following:

$$\begin{array}{lcl} [x] & = & x \\ [\mathsf{Type}_\ell] & = & \mathsf{Type}_\ell \\ [(x : \tau_1) \to \tau_2] & = & \begin{cases} ||(x : [\tau_1]) \to [\tau_2]|| & \text{if in the } \mathsf{Type}_0 \text{ universe} \\ (x : [\tau_1]) \to [\tau_2] & \text{otherwise} \end{cases} \\ [e_1 @ e_2] & = & \begin{cases} \mathsf{bind}\ [e_1]\ \lambda f . f @ [e_2] & \text{if in the } \mathsf{Type}_0 \text{ universe} \\ [e_1] @ [e_2] & \text{otherwise} \end{cases} \\ [\lambda x : \tau . e] & = & \begin{cases} |\lambda x : [\tau] . [e]| & \text{if in the } \mathsf{Type}_0 \text{ universe} \\ \lambda x : [\tau] . [e] & \text{otherwise} \end{cases} \end{array}$$

Sadly, this encoding is a bit too naïve to preserve types. While the monadic erasure axioms proposed above are rather enticing (and similar to those used in [5]) we will need to tweak them a bit in order to be able to refine the encoding into one that is type preserving.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant N^o 298311/2012 and RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

References

[1] Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):121–154, April 1991. doi:10.1017/S0956796800020025.

[2] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Certified Programs and Proofs*, page 182–194, 2017. doi:10.1145/3018610.3018620.

[3] Thierry Coquand and Gérard P. Huet. The calculus of constructions. Technical Report RR-0530, INRIA, 1986.

[4] Stefan Monnier and Nathaniel Bos. Is impredicativity implicitly implicit? In *Types for Proofs and Programs*, Leibniz International Proceedings in Informatics (LIPIcs), pages 9:1–9:19, 2019. doi:10.4230/LIPIcs.TYPES.2019.9.

[5] Arnaud Spiwack. Notes on axiomatising hurkens's paradox, 2015. URL: https://arxiv.org/abs/1507.04577.

[6] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: https://arxiv.org/abs/1308.0729.

[7] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. In *International Conference on Functional Programming*, page 1–29, 2017. doi:10.1145/3110275.

Semantics for two-dimensional type theory

Benedikt Ahrens[1], Paige Randall North[2], and Niels van der Weide[3]

[1] Delft University of Technology, The Netherlands
University of Birmingham, United Kingdom
B.P.Ahrens@tudelft.nl
[2] University of Pennsylvania, United States
pnorth@upenn.edu
[3] University of Birmingham, United Kingdom
nnmvdw@gmail.com

1 Introduction

In recent years, efforts have been made to develop *directed* type theory. Roughly, directed type theory should correspond to Martin-Löf type theory (MLTT) as ∞-categories correspond to ∞-groupoids. Besides theoretical interest in directed type theory, it is hoped that such a type theory can serve as a framework for synthetic directed homotopy theory and synthetic ∞-category theory. Applications of those, in turn, include reasoning about concurrent processes [2].

Several proposals for *syntax* for directed type theory have been given (reviewed in Section 2), but are ad-hoc and are not always semantically justified. The *semantic* aspects of directed type theory are particularly underdeveloped; a general notion of model of a directed type theory is still lacking.

In this work, we approach the development of directed type theory from the semantic side. We introduce *comprehension bicategories* as a suitable mathematical structure for higher-dimensional (directed) type theory. Comprehension bicategories capture several different specific mathematical structures that have previously been used to interpret higher-dimensional or directed type theory.

From comprehension bicategories, we extract the core syntax—judgment forms and structural inference rules—of a two-dimensional dependent type theory that can accommodate directed type theory. We also give a soundness proof of our structural rules. In future work, we will equip our syntax and semantics with a system of variances and type and term formers for directed type theory.

A preprint describing this work in more detail is available on the arXiv [1].

2 Related Work

We review only some related work here; an in-depth review of related work is given in [1].

In [7], Licata and Harper developed a two-dimensional dependent type theory with a judgment for *equivalences* $\Gamma \vdash \alpha : M \simeq_A N$ between terms $M, N : A$. These equivalences are postulated to have (strict) inverses. The authors give an interpretation of types as groupoids.

Licata and Harper [6] (see also [5, Chapter 7]) also designed a *directed* two-dimensional type theory and gave an interpretation for it in the strict 2-category of categories. Their syntax has a judgment for *substitutions* between contexts, written $\Gamma \vdash \theta : \Delta$, and *transformations* between parallel substitutions. An important aspect of their work is *variance* of contexts/types, built into the judgments.

Nuyts [9] attempts to generalize the treatment of variance by Licata and Harper, and designs a directed type theory with additional variances, in particular, isovariance and invariance. Nuyts does not provide any interpretation of their syntax, and thus no proof of (relative) consistency.

North [8] develops a type former for *directed* types of morphisms, resulting in a typal higher-dimensional directed type theory based on the judgments of MLTT. The model given by North is in the 2-category of categories, similar to Licata and Harper's [6].

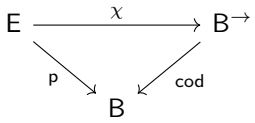
Shulman, in unfinished work [10], aims to develop 2-categorical logic, including a two-dimensional notion of topos and a suitable internal language for such toposes. Our work is similar to Shulman's in the sense that both start from a (bi)categorical notion and extract a language from it, with the goal of developing a precise correspondence between extensions of the syntax and additional structure on the semantics.

Garner [3] studies a typal two-dimensional type theory in the style of Martin-Löf. They add rules that turn any identity type into a discrete type, effectively "truncating" intensional Martin-Löf type theory at 1-types. Garner defines a notion of two-dimensional model based on (strict) *comprehension 2-categories*. Exploiting the restriction to 1-truncated types, they then give a sound and complete interpretation of their two-dimensional type theory in any model.

3 Details

We introduce a notion of "model" of two-dimensional type theory that is a quite straightforward generalization of the 1-categorical comprehension categories introduced by Jacobs [4].

Definition 1. A **comprehension bicategory** is given by a *strictly* commuting diagram of pseudofunctors



where p is equipped with a global cleaving and a local opcleaving (modelling substitution in a suitable sense), opcartesian 2-cells of p are preserved under left and right whiskering, and χ preserves cartesian 1-cells and opcartesian 2-cells.

Examples of comprehension bicategories are plentiful; in particular, taking $\mathsf{B} \equiv \mathsf{Cat}$ and $\mathsf{E} \equiv \mathsf{OpFib}$ yields a structure that is similar to, but crucially not the same as, the structure in which Licata and Harper's interpretation [6] takes place. (A comparison is given in [1, §7.6].)

From the definition of comprehension bicategories we extract a core type theory called BTT. We then prove

Theorem 1 (Soundness)**.** *We can interpret* BTT *in any comprehension bicategory.*

Our type theory can be simplified in lockstep with the categorical structure in which it is interpreted; we give various pairs of simplifications in [1, §7.5].

4 Conclusion

Our work is very general; it allows for the modelling of the structural rules of previous suggestions for directed type theory. Furthermore, it can be used as a framework for defining and studying more specialized syntax and semantics, in lockstep.

In separate work we are going to extend our structural rules with variances and a suitable hom-type former à la North [8].

References

[1] Benedikt Ahrens, Paige Randall North, and Niels van der Weide. Semantics for two-dimensional type theory, 2022. https://arxiv.org/abs/2201.10662v1.

[2] Lisbeth Fajstrup, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. *Directed Algebraic Topology and Concurrency*. Springer, 2016.

[3] Richard Garner. Two-dimensional models of type theory. *Math. Struct. Comput. Sci.*, 19(4):687–736, 2009.

[4] Bart Jacobs. Comprehension categories and the semantics of type dependency. *Theor. Comput. Sci.*, 107(2):169–207, 1993.

[5] Daniel R. Licata. *Dependently Typed Programming with Domain-Specific Logics*. PhD thesis, USA, 2011. AAI3476124.

[6] Daniel R. Licata and Robert Harper. 2-dimensional directed type theory. In Michael W. Mislove and Joël Ouaknine, editors, *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011*, volume 276 of *Electronic Notes in Theoretical Computer Science*, pages 263–289. Elsevier, 2011.

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

[8] Paige Randall North. Towards a directed homotopy type theory. In Barbara König, editor, *Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2019, London, UK, June 4-7, 2019*, volume 347 of *Electronic Notes in Theoretical Computer Science*, pages 223–239. Elsevier, 2019.

[9] Andreas Nuyts. Towards a directed homotopy type theory based on 4 kinds of variance. Master's thesis, KU Leuven, 2015.

[10] Michael Shulman. 2-categorical logic. https://ncatlab.org/michaelshulman/show/2-categorical+logic.

Session Type Systems Compared: The Case of Deadlock Freedom

Ornela Dardha[1]*and Jorge A. Pérez[2]

[1] School of Computing Science, University of Glasgow, UK
ornela.dardha@glasgow.ac.uk
[2] University of Groningen, The Netherlands
j.a.perez@rug.nl

Abstract

This note summarizes our recent work [6], in which we develop a comparative study of different type systems for message-passing processes that guarantee deadlock freedom. We actually compare two classes of deadlock-free typed processes, denoted $\mathcal{L}$ and $\mathcal{K}$. The class $\mathcal{L}$ stands out for its canonicity: it results from Curry-Howard interpretations of classical linear logic propositions as session types. The class $\mathcal{K}$, obtained by encoding session types into Kobayashi's linear types with usages, includes processes not typable in other type systems. We show that $\mathcal{L}$ is strictly included in $\mathcal{K}$, and identify the precise conditions under which they coincide. We also provide two type-preserving translations of processes in $\mathcal{K}$ into processes in $\mathcal{L}$.

1 Introduction

We are interested in formally relating different type systems for concurrent processes specified in the π-calculus [11]. Our focus is on *session-based concurrency*, the model of concurrency captured by session types. Session types promote a type-based approach to communication correctness: dialogues between participants are structured into *sessions*, basic communication units; descriptions of interaction sequences are then abstracted as session types which are checked against process specifications. In session-based concurrency, types enforce correct communications through different safety and liveness properties. Two basic (and intertwined) correctness properties are *communication safety* and *session fidelity*. A desirable property for safe processes is that they should never "get stuck", namely the property of *deadlock freedom*.

In our recent work [6], we present the *first formal comparison* between different type systems for the π-calculus that enforce properties related to (dead)lock freedom. More concretely, we compare $\mathcal{L}$ and $\mathcal{K}$, two salient classes of deadlock-free (session) typed processes, which are induced by different type systems:

- The class $\mathcal{L}$ contains session processes that are well-typed under the Curry-Howard correspondence between (classical) linear logic propositions and session types [1, 2, 13]. Requiring well-typedness suffices, because the type system derived from such a correspondence simultaneously ensures communication safety, session fidelity, and deadlock freedom.
- The class $\mathcal{K}$ contains session processes that enjoy communication safety and session fidelity (as ensured by the type system of Vasconcelos [12]) as well as satisfy deadlock freedom. This class of processes is defined indirectly, by combining Kobayashi's linear type system based on usages [7, 9, 10] with encodability results by Dardha et al. [5].

*Work supported by the EU H2020 MSCA RISE project BehAPI and by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

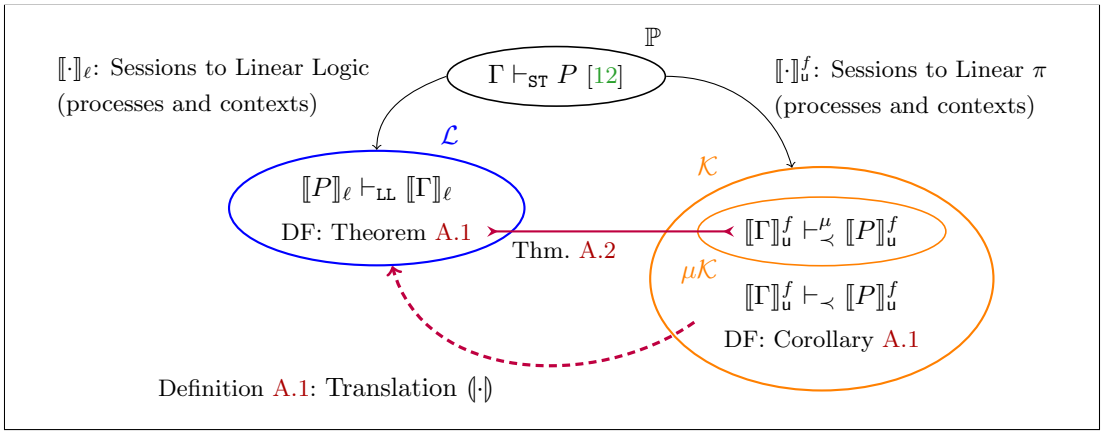


Figure 1: Summary of type systems, languages with deadlock freedom (DF), and encodings between them (indicated by solid black arrows). Main results in our recent work [6] are denoted by purple lines: our separation result, based on the coincidence of $\mathcal{L}$ and $\mu\mathcal{K}$ is indicated by the solid line with reversed arrowheads; our unifying result is indicated by the dashed arrow.

2 Contributions

Our work develops two kinds of technical results, summarized by Figure 1. On the one hand, we give results that *separate* the classes $\mathcal{L}$ and $\mathcal{K}$ by precisely characterizing the fundamental differences between them; on the other hand, we precisely explain how to *unify* these classes by showing how their differences can be overcome to translate processes in $\mathcal{K}$ into processes into $\mathcal{L}$. More in details:

- To *separate* $\mathcal{L}$ from $\mathcal{K}$, we define $\mu\mathcal{K}$: a sub-class of $\mathcal{K}$ whose definition internalizes the key aspects of the Curry-Howard interpretations of session types. In particular, $\mu\mathcal{K}$ adopts the principle of "composition plus hiding", a distinguishing feature of the interpretations in [1, 13], by which the cut rule in linear logic is interpreted as the concurrent cooperation between two processes that interact on *exactly one* session channel.

  We show that $\mathcal{L}$ and $\mu\mathcal{K}$ coincide (Theorem A.2). This gives us a separation result: there are deadlock-free session processes that *cannot* be typed by systems derived from the Curry-Howard interpretation of session types [1, 2, 13], but that are admitted as typable by the (indirect) approach of [3, 4].

- To *unify* $\mathcal{L}$ and $\mathcal{K}$, we define two *translations* of processes in $\mathcal{K}$ into processes in $\mathcal{L}$ (Definition A.1). Intuitively, because the difference between $\mathcal{L}$ and $\mathcal{K}$ lies in the forms of parallel composition they admit (restricted in $\mathcal{L}$, liberal in $\mathcal{K}$), it is natural to transform a process in $\mathcal{K}$ into another, more parallel process in $\mathcal{L}$. In essence, the first translation, denoted $(\!|\cdot|\!)$ (Definition A.1), exploits type information to replace sequential prefixes with representative parallel components; the second translation refines this idea by considering *value dependencies*, i.e., causal dependencies between independent sessions not captured by types. We detail the first translation, which satisfies type-preservation and operational correspondence properties (Theorems A.3 and A.4).

References

[1] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[2] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

[3] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec*, volume 8459 of *LNCS*, pages 49–64. Springer, 2014.

[4] Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*, volume 7 of *Atlantis Studies in Computing*. Springer / Atlantis Press, 2016.

[5] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM, 2012.

[6] Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *J. Log. Algebraic Methods Program.*, 124:100717, 2022.

[7] Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.

[8] Naoki Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads: From Panacea to Foundational Support—Papers from the 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of the United Nations University*, volume 2757 of *LNCS*, pages 439–453. Springer, 2002.

[9] Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR 2006*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006. Full version available at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/papers/concur2006-full.pdf`.

[10] Naoki Kobayashi. Type systems for concurrent programs. Extended version of [8], Tohoku University. `www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`, 2007.

[11] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[12] Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.

[13] Philip Wadler. Propositions as sessions. In *ICFP'12*, pages 273–286, 2012.

A Results

A.1 DF in Sessions, Linear π and Linear Logic

For any P, define $live(P)$ if and only if $P \equiv (\boldsymbol{\nu}\widetilde{n})(\pi.Q \mid R)$, where π is an input, output, selection, or branching prefix.

Theorem A.1 (Deadlock Freedom)**.** *If $P \vdash_{\mathtt{LL}} \cdot$ and $live(P)$ then $P \longrightarrow Q$, for some Q.*

The following result states deadlock freedom by encodability, following [3].

Corollary A.1. *Let $\vdash_{\mathtt{ST}} P$ be a session process. If $\vdash_{\prec} [\![P]\!]^{f}_{\mathsf{u}}$ is deadlock-free then P is deadlock-free.*

A.2 Relating $\mathcal{L}$, $\mu\mathcal{K}$ and $\mathcal{K}$

Theorem A.2. *$\mathcal{L} = \mu\mathcal{K}$.*

Definition A.1 (Translation into $\mathcal{L}$)**.** *Let P be such that $\Gamma \vdash_{\mathtt{ST}} P$ and $P \in \mathcal{K}$. The set of $\mathcal{L}$ processes $(\!|\Gamma \vdash_{\mathtt{ST}} P|\!)$ is defined in Figure 2.*

$$
\begin{aligned}
(\!|\Gamma^{\mathsf{un}} \vdash_{\mathtt{ST}} \mathbf{0}|\!) &\triangleq \{\mathbf{0}\} \\
(\!|\Gamma, x : !T.S, v : T \vdash_{\mathtt{ST}} \overline{x}\langle v\rangle.P'|\!) &\triangleq \{\overline{x}(z).([v \leftrightarrow z] \mid Q) \ : \ Q \in (\!|\Gamma, x : S \vdash_{\mathtt{ST}} P'|\!)\} \\
(\!|\Gamma_1, \Gamma_2, x : !T.S \vdash_{\mathtt{ST}} (\boldsymbol{\nu} zy)\overline{x}\langle y\rangle.(P_1 \mid P_2)|\!) &\triangleq \\
&\{\overline{x}(y).(Q_1 \mid Q_2) \ : \ Q_1 \in (\!|\Gamma_1, z : \overline{T} \vdash_{\mathtt{ST}} P_1|\!) \wedge Q_2 \in (\!|\Gamma_2, x : S \vdash_{\mathtt{ST}} P_2|\!)\} \\
(\!|\Gamma, x : ?T.S \vdash_{\mathtt{ST}} x(y : T).P'|\!) &\triangleq \{x(y).Q \ : \ Q \in (\!|\Gamma, x : S, y : T \vdash_{\mathtt{ST}} P'|\!)\} \\
(\!|\Gamma, x : \oplus\{l_i : S_i\}_{i\in I} \vdash_{\mathtt{ST}} x \triangleleft l_j.P'|\!) &\triangleq \{x \triangleleft l_j.Q \ : \ Q \in (\!|\Gamma, x : S_j \vdash_{\mathtt{ST}} P'|\!)\} \\
(\!|\Gamma, x : \&\{l_i : S_i\}_{i\in I} \vdash_{\mathtt{ST}} x \triangleright \{l_i : P_i\}_{i\in I}|\!) &\triangleq \{x \triangleright \{l_i : Q_i\}_{i\in I} \ : \ Q_i \in (\!|\Gamma, x : S_i \vdash_{\mathtt{ST}} P_i|\!)\} \\
(\!|\Gamma_1, \widetilde{[x : S]} \star \Gamma_2, \widetilde{[y : T]} \vdash_{\mathtt{ST}} (\boldsymbol{\nu}\widetilde{x}\widetilde{y} : \widetilde{S})(P_1 \mid P_2)|\!) &\triangleq \\
&\{C_1[Q_1] \mid G_2 \ : \ Q_1 \in (\!|\Gamma_1, \widetilde{x : S} \vdash_{\mathtt{ST}} P_1|\!),\ C_1 \in \mathcal{C}_{\widetilde{x:T}},\ G_2 \in \langle\!|\Gamma_2|\!\rangle\} \\
&\qquad \cup \\
&\{G_1 \mid C_2[Q_2] \ : \ Q_2 \in (\!|\Gamma_2, \widetilde{y : T} \vdash_{\mathtt{ST}} P_2|\!),\ C_2 \in \mathcal{C}_{\widetilde{y:S}},\ G_1 \in \langle\!|\Gamma_1|\!\rangle\}
\end{aligned}
$$

Figure 2: Translation $(\!|\cdot|\!)$ (cf. Definition A.1).

We present two important results about our translation. First, it is type preserving, up to the encoding of types:

Theorem A.3 (The Translation $(\!|\cdot|\!)$ is Type Preserving)**.** *Let $\Gamma \vdash_{\mathtt{ST}} P$. Then, for all $Q \in (\!|\Gamma \vdash_{\mathtt{ST}} P|\!)$, we have that $Q \vdash_{\mathtt{LL}} [\![\Gamma]\!]_\ell$.*

Definition A.2 (Parallelization Relation)**.** *Let P and Q be processes such that $P, Q \vdash_{\mathtt{LL}} \Gamma$. We write $P \doteqdot Q$ if and only if there exist processes P_1, P_2, Q_1, Q_2 and contexts Γ_1, Γ_2 such that the following hold:*

$$P = P_1 \mid P_2 \qquad Q = Q_1 \mid Q_2 \qquad P_1, Q_1 \vdash_{\mathtt{LL}} \Gamma_1 \qquad P_2, Q_2 \vdash_{\mathtt{LL}} \Gamma_2 \qquad \Gamma = \Gamma_1, \Gamma_2$$

By definition, the relation $\doteqdot$ is reflexive. We may now state:

Theorem A.4 (Operational Correspondence for $(\!|\cdot|\!)$)**.** *Let P be such that $\Gamma \vdash_{\mathtt{ST}} P$ for some typing context Γ. Then, we have:*

1. *If $P \to P'$, then for all $Q \in (\!|\Gamma \vdash_{\mathtt{ST}} P|\!)$ there exist Q', R such that $Q \to\!\!\!\!\hookrightarrow Q'$, $Q' \doteqdot R$, and $R \in (\!|\Gamma \vdash_{\mathtt{ST}} P'|\!)$.*
2. *If $Q \in (\!|\Gamma \vdash_{\mathtt{ST}} P|\!)$, such that $P \in \mathcal{K}$, and $Q \to\!\!\!\!\hookrightarrow Q'$, then there exist P', R such that $P \to P'$, $Q' \doteqdot R$, and $R \in (\!|\Gamma \vdash_{\mathtt{ST}} P'|\!)$.*

Sikkel: Multimode Simple Type Theory as an Agda Library

Joris Ceulemans, Andreas Nuyts, and Dominique Devriese

imec-DistriNet, Department of Computer Science, KU Leuven, Belgium

Many variants of type theory extend a basic theory with additional primitives or properties to enable constructions or proofs that would be harder or impossible to do in the original theory. Examples of such extensions include guarded recursion [7, 13], parametricity [2, 3, 4, 19, 18, 8], univalence [5, 9, 11], directed type theory [21, 22] and nominal reasoning [20]. The question addressed in this abstract is how these extended systems can be used in existing dependently typed programming languages like Agda or Coq. We present Sikkel[1]: an Agda library that allows users to work in a class of extended (simple) type theories called multimode or multimodal type theories [16, 13], which are parametrized by a mode theory that specifies new primitive type constructors called modalities (but non-modal primitives can be easily added to Sikkel as well). Such modalities are not only useful in the object theory, but they also allow for an elegant translation of Sikkel programs to Agda programs. Fig. 1 shows Sikkel's different components, which will be discussed in the following sections.

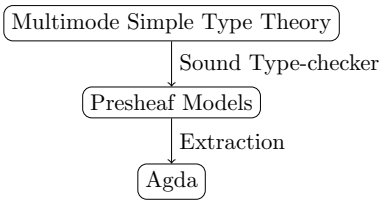


Figure 1: Sikkel's architecture.

This abstract is based on a full paper, which was presented at MSFP 2022 [10].

Multimode Simple Type Theory (MSTT) The upper layer of Fig. 1 represents MSTT, which is essentially Gratzer et al.'s MTT [13] restricted to simple types.[2] Our implementation of MSTT is parametrized by a ModeTheory record, which specifies an Agda type ModeExpr of modes, a type family ModalityExpr of modalities (indexed by two modes), as well as a unit modality, modality composition and a type of 2-cells (i.e. a mode theory is basically a strict 2-category). All MSTT contexts, types and terms live at a certain mode and are represented in Sikkel as values of data types CtxExpr, TyExpr and TmExpr indexed by a mode. Sikkel's MSTT syntax is extrinsically typed, so TmExpr is not indexed by a context or type. There is built-in support for booleans, natural numbers, function types and product types. Every modality μ from mode m to n gives rise to a modal type former, transforming a type T at mode m to $\langle\ \mu\ \mid\ T\ \rangle$ at mode n, and a lock (left adjoint) on contexts, transforming a context Γ at mode n to $\Gamma\ ,\mathsf{lock}\langle\ \mu\ \rangle$ at mode m. Variables are represented by strings[3] and always appear in the context under a modality. The following are 3 of the more interesting MSTT typing rules.

$$\frac{\alpha \in \mu \Rightarrow \mathsf{locks}(\Delta) \qquad x \notin \Delta}{\Gamma\ ,\ \mu\ \mid\ x \in T\ ,\Delta \vdash \mathsf{var}\ x\ \alpha : T}\ \textsc{Var} \qquad \frac{\Gamma\ ,\ \mu\ \mid\ x \in T \vdash s : S}{\Gamma \vdash \mathsf{lam}[\ \mu\ \mid\ x \in T\ ]\ s : \langle\ \mu\ \mid\ T\ \rangle \Rrightarrow S}\ \textsc{Lam} \qquad \frac{\Gamma\ ,\mathsf{lock}\langle\ \mu\ \rangle \vdash t : T}{\Gamma \vdash \mathsf{mod}\langle\ \mu\ \rangle\ t : \langle\ \mu\ \mid\ T\ \rangle}\ \textsc{ModIntro}$$

A variable x that is in the context under modality μ can be used to construct a term, as long as there is a 2-cell from μ to the composite of all modalities that appear in locks to the right of x (Var). In many cases, these two modalities are the same so we can replace α with the trivial 2-cell which we abbreviate as svar x. Lambda abstraction produces a modal function by extending the context with a variable under the same modality (Lam). The introduction rule for modalities (ModIntro) shows that we can construct a term of type $\langle\ \mu\ \mid\ T\ \rangle$ whenever we have a term of type T in the context locked with μ.

[1] Available at https://github.com/JorisCeulemans/sikkel/releases/tag/v1.0.

[2] So just like MTT, MSTT does not support substructural modal systems such as [17] or [1].

[3] The strings are resolved to de Bruijn indices when going from the upper to the middle layer in Fig. 1. MSTT does not specify an equational theory for terms or reduction of terms.

Fig. 2 shows a simple example of a modal program written in Sikkel (the symbol · is MSTT function application). It has type $\langle\ \mu \mid T \Rrightarrow S\ \rangle \Rrightarrow \langle\ \mu \mid T\ \rangle \Rrightarrow \langle\ \mu \mid S\ \rangle$ and proves that every modality satisfies the K axiom (or, in other words, is an applicative functor).

```
applicative : ModalityExpr m n → TmExpr n
applicative μ = lam[ μ | "f" ∈ T ⇛ S ]
  lam[ μ | "t" ∈ T ] mod⟨ μ ⟩ (svar "f" · svar "t")
```

Figure 2: Example of a modal program in Sikkel.

Presheaf Models & Sound Type-checker MSTT does not have interesting computational behavior in the sense that there is no reduction for terms. Sikkel's intended use is that certain MSTT terms will be interpreted as Agda terms. However, in an off-the-shelf version of Agda this is not immediately possible for many of the modal type and term formers. For this reason, there is a middle layer in Fig. 1, representing a formalization of presheaf models of type theory in Agda, which is essentially a shallow embedding of MSTT [23]. A presheaf model is parametrized by a base category, different modes will correspond to different base categories and they determine which new type and term formers are implementable in the model. Modalities are interpreted as dependent right adjunctions (DRAs) [6] and allow to transfer semantic types and terms between different base categories. Our formalization follows the general construction by Hofmann [15] and is structured as an internal Category with Families (CwF) [12]. Although MSTT is simply-typed, Sikkel's semantic layer already anticipates the addition of dependent types and semantic types may depend on variables.

The bridge between Sikkel's first two layers is a type checker which is sound by construction in the sense that it will either refute a judgment or accept it *and* interpret it in the presheaf model. In order for this to be possible, a Sikkel user implementing a new type theory must provide interpretations of modes as base categories, modalities as DRAs and two-cells as certain natural transformations. Furthermore, if any non-modal type or term formers are added, the user must specify how they should be type-checked and interpreted.

Extraction The presheaf model over the trivial base category corresponds to the standard set model of type theory. We make use of this fact to extract semantic terms in this model to the meta-level (i.e. to Agda terms), as shown at the bottom of Fig. 1.[4] However, the interpretation of some types (e.g. function types) in the model is only *isomorphic* to their intended meaning. This is why Sikkel provides a type class Extractable for semantic types, instances of which declare the intended translated Agda type and provide a way to apply the isomorphism.

Applications We implemented two applications in Sikkel: guarded recursive type theory and a restricted form of parametricity. With guarded recursion, we can write definitions of infinite streams that would not be accepted by Agda's termination checker. The extraction mechanism allows us to interpret these definitions as ordinary Agda streams. For parametricity, we demonstrate how a function definition can be interpreted with two different representations for an abstract type, and how parametricity allows us to relate the two resulting definitions.

Future Work Eventually, we plan to extend Sikkel with support for dependent types. However, the interpretation of a dependently typed syntax turns out to be a significant challenge (especially to satisfy Agda's termination checker). Furthermore, the addition of a Hofmann-Streicher universe to our presheaf model is non-trivial. Therefore, we first intend to equip Sikkel with an extensible program logic (in the same spirit as the Edinburgh Logical Framework [14]), orthogonal to the different layers in Fig. 1, which will allow a programmer to prove properties of functions written in Sikkel and translate the proofs to Agda proofs of the extracted Agda functions. We also plan to study more examples of multimode type theories as applications of Sikkel, with an ongoing exploration of nominal type theory [20].

[4] Semantic terms over other base categories must first be transferred by applying a modality.

Acknowledgements Joris Ceulemans and Andreas Nuyts hold a PhD Fellowship and a Postdoctoral Fellowship, respectively, from the Research Foundation – Flanders (FWO). This work was partially supported by a research project of the Research Foundation - Flanders (FWO).

References

[1] Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020. doi:10.1145/3408972.

[2] Robert Atkey. Relational parametricity for higher kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46–61, 2012. doi:10.4230/LIPIcs.CSL.2012.46.

[3] Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *Principles of Programming Languages*, 2014. doi:10.1145/2535838.2535852.

[4] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electron. Notes in Theor. Comput. Sci.*, 319:67 – 82, 2015. doi:10.1016/j.entcs.2015.12.006.

[5] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, Dagstuhl, Germany, 2014. URL: http://drops.dagstuhl.de/opus/volltexte/2014/4628, doi:10.4230/LIPIcs.TYPES.2013.107.

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

[7] Ales Bizjak and Rasmus Ejlers Møgelberg. Denotational semantics for guarded dependent type theory. *Math. Struct. Comput. Sci.*, 30(4):342–378, 2020. doi:10.1017/S0960129520000080.

[8] Evan Cavallo and Robert Harper. Internal parametricity for cubical type theory. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*, pages 13:1–13:17, 2020. doi:10.4230/LIPIcs.CSL.2020.13.

[9] Evan Cavallo, Anders Mörtberg, and Andrew W Swan. Unifying cubical models of univalent type theory. In Maribel Fernández and Anca Muscholl, editors, *Computer Science Logic (CSL 2020)*, volume 152 of *LIPIcs*, pages 14:1–14:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/11657, doi:10.4230/LIPIcs.CSL.2020.14.

[10] Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. Sikkel: Multimode simple type theory as an Agda library. In *Proceedings of the Ninth Workshop on Mathematically Structured Functional Programming*, pages 93–112, Munich, Germany, 2022. URL: http://eptcs.web.cse.unsw.edu.au/paper.cgi?MSFP2022.5.

[11] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017. URL: http://www.cse.chalmers.se/~simonhu/papers/cubicaltt.pdf.

[12] 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. doi:10.1007/3-540-61780-9_66.

[13] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021. URL: https://lmcs.episciences.org/7713, doi:10.46298/lmcs-17(3:11)2021.

[14] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, jan 1993. doi:10.1145/138027.138060.

[15] Martin Hofmann. *Syntax and Semantics of Dependent Types*, chapter 4, pages 79–130. Cambridge University Press, 1997.

[16] Daniel R. Licata and Michael Shulman. Adjoint logic with a 2-category of modes. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science - International Symposium, LFCS 2016, Deerfield Beach, FL, USA, January 4-7, 2016. Proceedings*, volume 9537 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2016. doi:10.1007/978-3-319-27683-0_16.

[17] Daniel R. Licata, Michael Shulman, and Mitchell Riley. A fibrational framework for substructural and modal logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7740, doi:10.4230/LIPIcs.FSCD.2017.25.

[18] Andreas Nuyts and Dominique Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 779–788, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209119.

[19] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110276.

[20] Andrew M. Pitts, Justus Matthiesen, and Jasper Derikx. A dependent type theory with abstractable names. *Electronic Notes in Theoretical Computer Science*, 312:19 – 50, 2015. Ninth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2014). URL: http://www.sciencedirect.com/science/article/pii/S1571066115000079, doi:10.1016/j.entcs.2015.04.003.

[21] E. Riehl and M. Shulman. A type theory for synthetic ∞-categories. *ArXiv e-prints*, May 2017. arXiv:1705.07442.

[22] Matthew Z. Weaver and Daniel R. Licata. A constructive model of directed univalence in bicubical sets. 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 915–928. ACM, 2020. doi:10.1145/3373718.3394794.

[23] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 305–320, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-30142-4_22.

Small inversions for smaller inversions

Jean-François Monin

Abstract

We describe recent improvements on *small inversions*, a technique presented earlier as a possible alternative to Coq standard inversion.

Many proofs relying on inductive definitions require so-called *inversion* steps in order to exploit the information contained in a hypothesis $H : Ra_0 \dots a_n$, where R is a dependent inductive relation (or type) applied to actual parameters $a_0 \dots a_n$. In Coq, such steps are usually performed using a powerful tactic called `inversion`. In previous work [MS13], we proposed an alternative lightweight approach, which is not automated (in contrast with Coq `inversion`) but provides a better understanding on what happens, a full control on the proof script and smaller proof terms. In particular, no additional (proof of) equalities are introduced. Moreover, there are situations involving dependent types where standard `inversion` fails whereas small inversion succeeds. In all approaches, inversion is essentially a complex dependent pattern matching on H. In [MS13], this is handled using auxiliary functions.

However, we discovered later that standard `inversion` has an additional important advantage on the previous version of small inversions: it provides syntactically strict subterms of H which can directly be used in recursive calls of a fixpoint definition. In the Braga method designed with D. Larchey-Wendling [LWM18, LM21], we showed how clear and explicit subterms of H can be recovered using projections π_R defined with a dependent pattern matching similar to small inversions – let us call them *smaller inversions*.

We present here an improvement on small inversions, where auxiliary functions are replaced with auxiliary inductive types which are easier to understand and to use. The new small inversion is more powerful: it can handle goals involving terms with occurrences of H. Such goals naturally arise in direct proofs of partial correctness properties of functions – for instance, but not only, fixpoints obtained by the Braga method. Standard `inversion` turns out to be very often unusable there.

As a foretaste, consider a *reference* function for OCaml `fold_left` – efficiency is then irrelevant here – honestly defined by a right to left traversal of its list argument. To this effect we introduce an auxiliary *non-recursive* dependent data type `rl` l with two constructors: `Nilr` of type `rl` [] – reflecting the empty list – and `Consr` of type `rl` $(u +\!: z)$ – where $u +\!: z$ is the catenation of a list u and a single element z. Following the Braga method, we first define an inductive domain D_{list} for termination certificates. Here D_{list} contains `Nilr`, as well as `Consr` u z whenever `l2r` u, the reflection of u, is itself in D_{list}. Given $d : D_{list}$ (`Consr` u z) we then define the projection $\pi\ d$ which provides its structurally smaller component of type D_{list} (`l2r` u), allowing us to easily define fixpoints such as `foldl_ref` below, where b_0 and f are respectively an initial value and a function to be folded, and `rew` d is an administrative rewriting step transforming `l2r` $(u +\!: z)$ into `Consr` u z.

```
Fixpoint foldl_ref l (d: D_list (l2r l)): B :=
  match l2r l in rl l return D_list (l2r l) → B with
  | Nilr      => λ d, b0
  | Consr u z => λ d, f (foldl_ref u (π (rew d))) z
  end d.
```

Reasoning on such functions commonly requires inversion steps on d. For instance we would like to prove that the actual standard tail-recursive algorithm returns the same result as

`foldl_ref`. But we already get an issue with a much more elementary fact, stating that for any $d : D_{list}$ `Nilr`, we have `foldl_ref` [] $d = b_0$: this turns out to be out of reach of Coq standard inversion. In [LM21], this issue is circumvented by replacing the former definition of `foldl_ref` by an enriched program which returns an inhabitant of $\{b : B \mid G_{foldl}\ l\ b\}$ instead of just B, where G_{foldl} is a suitable characteristic relation. With our small inversion described below, we can directly reason on `foldl_ref` as defined above. More details and additional examples are available at https://www-verimag.imag.fr/~monin/Proof/Small_inversions/2022 .

For a more general situation, consider an inductive relation $R : T_0 \to T_1 \ldots \to T_n \to Sort$, where $Sort$ is a sort (e.g., Prop or Type). Whatever the technology to be used, the key point is that inversion makes sense when:

- at least one type among $T_0, T_1 \ldots T_n$, say T_0, is itself an inductive type; without loss of generality we consider here that there is exactly one such type; below we write $\mathbf{T}$ for $T_1 \ldots T_n$;
- in the hypothesis H to be inverted, the corresponding actual parameter $a_0 : T_0$ has a specialized shape σ, corresponding to a pattern $C\ args$ starting with a constructor C of T_0 (in many cases, $args$ are just variables).

In general, only a subset of the constructors of R are compatible with the shape of a_0. Inversion then proceeds by *simultaneous* pattern-matching on H and a_0, in order to select the relevant cases of R.

We proceed as follows. For each shape of interest σ we derive from the definition of T_0 an inductive specialized version $T_{0\sigma}$ of T_0. $T_{0\sigma}$ is a copy-paste of the relevant (compatible with σ) constructors of T_0, with appropriate modifications: the variables $x_1 \ldots x_\sigma$ of σ become parameters of $T_{0\sigma}$; the type of $T_{0\sigma}\ x_1 \ldots x_\sigma$ is $\forall\, \mathbf{a} : \mathbf{T}, R\, \sigma\, \mathbf{a} \to Sort$ (it is empty for absurd cases). We then define, by dependent pattern matching on r, the function $R_{inv}\ y_0\ \mathbf{y}$ of type $\forall\, r : R\, y_0\, \mathbf{y}$, (`match` y_0 `with` $\ldots \mid \sigma_i \Rightarrow T_{0\sigma_i}\ x_1 \ldots x_{\sigma_i} \mid \ldots$ `end`) r.
The main argument of R_{inv} is r (its other arguments $y_0\ \mathbf{y}$ will be left implicit). An obvious requirement on the shapes σ_i occuring in the above pattern matching is that they cover T_0. Inverting H is then just a pattern matching on $R_{inv}\ H$, whose type reduces to the relevant $T_{0\sigma_i}$. Possible occurrences of H in the goal are correctly dealt with for free thanks to the additional argument r of $T_{0\sigma}$.

In this version, the components of H are not considered as subterms of H because they are repackaged in a constructor of $T_{0\sigma}$. In the Braga method, where the subterm property is needed, an argument of type $R\, y_0\, \mathbf{y}$ can be added to $T_{0\sigma}$ and its final argument uses of $\pi_R\ \sigma$ instead of σ. Even if the sort of T_0 is Prop, we can then obtain a fully general recursion principle T_0`_rect` – an improvement on [LM21] which is limited to proof irrelevant statements.

References

[LM21] Dominique Larchey-Wendling and Jean-François Monin. *The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq*, chapter 8, pages 305–386. World Scientific, September 2021.

[LWM18] Dominique Larchey-Wendling and Jean-François Monin. Simulating Induction-Recursion for Partial Algorithms. In *24th International Conference on Types for Proofs and Programs, TYPES 2018*, Braga, Portugal, June 2018.

[MS13] Jean-François Monin and Xiaomu Shi. Handcrafted Inversions Made Operational on Operational Semantics. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 338–353. Springer, 2013.

Strong, Synthetic, and Computational Proofs of Gödel's First Incompleteness Theorem

Benjamin Peters and Dominik Kirst

Saarland University, Saarbrücken, Germany

There is a simple folklore proof of Gödel's first incompleteness theorem (G1) by Kleene using computability theory and undecidability of the halting problem [8]. As opposed to Gödel's original proof [6], which directly arithmetizes provability, and Rosser's improvement on this result by modifying the provability predicate [17], Kleene's proof is much easier to spell out in detail, only relying on basic results in computability theory [9].

Constructive logics are a useful tool for formalizing results in computability theory because in such logics usually all definable functions are computable, avoiding the need to argue via a concrete model of computation. They therefore appear to provide an elegant way to formalize G1 when combined with Kleene's folklore proof [7]. However, Kleene's proof only shows a considerably weaker statement: It only works for sound as opposed to just consistent formal systems, and does not construct an independent sentence.

Nevertheless, Kleene did find a way to fix these weaknesses using Rosser's trick [10, 9]. However, this result is much less well-known, as evidenced by [1, 20].

We first outline how to formalize both the folklore and the strengthened versions of Kleene's incompleteness proofs for abstract formal systems in the calculus of inductive constructions (CIC) [3, 14]. To do this synthetically, we assume the axiom of Church's thesis [12, 19, 4], internalizing the fact that all constructively definable functions are computable. Secondly, we instantiate these proofs with a concrete presentation of first-order logic using Rosser's trick.

Most of the results presented here have been mechanized using the Coq proof assistant [18].

Synthetic computability. We are using *synthetic computability theory* [15, 2, 5] to formalize our results without directly working with a concrete model of computation. We write $X^?$ for the option type $X + \mathbb{1}$, containing values $^\circ x$ and a none value. We say that a predicate $P : X \to \mathbb{P}$ is enumerable if there is a function $f : \mathbb{N} \to X^?$ such that $\forall x.\, Px \leftrightarrow \exists k.\, fk = {}^\circ x$, and decidable if there is a function $f : X \to \mathbb{B}$ such that $\forall x.\, Px \leftrightarrow fx = tt$. We also work with a type of partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$. It can for example be realized using step indexed functions $\mathbb{N} \to \mathbb{N} \to \mathbb{N}^?$. We write $fx \triangleright y$ if fx halts and evaluates to y.

Weak G1. Our abstract notion of a formal system consists of an enumerable and discrete type of sentences S, an enumerable provability predicate $\vdash : S \to \mathbb{P}$, and a negation function $\neg : S \to S$ such that $\vdash$ is consistent: $\forall s. \neg(\vdash s \wedge \vdash \neg s)$. We call a formal system complete if $\forall s. \vdash s \vee \vdash \neg s$. Note that in a complete formal system, provability is decidable.

We say that a formal system weakly represents a predicate $P : \mathbb{N} \to \mathbb{P}$ if there is a representation function $R_P : \mathbb{N} \to S$ such that $\forall x.\, Px \leftrightarrow \vdash R_P\, x$.

Assume that there is a formal system that is complete and weakly represents the halting problem H for some model of computation. Now, H is decidable, because $\lambda x. \vdash R_H\, x$ is decidable, since the formal system is complete. This is the folklore proof of G1, as mechanized in [7].

There are multiple ways in which we strengthen this result, following Kleene:

- Instead of decidability of the halting problem, we derive falsity from completeness.

- There are unsound (but consistent) formal systems that do not weakly represent H because the direction from right to left requires a form of soundness. We show incompleteness even for such formal systems by modifying the representability property required.
- We explicitly construct an independent sentence, that is, we show $\exists s. \nvdash s \land \nvdash \neg s$.

Unfortunately, the above notions from synthetic computability theory are not strong enough to obtain these results without directly working with a concrete model of computation.

Church's thesis. We can however internalize the notion that all definable functions are computable by assuming a formulation of the axiom of Church's thesis (CT) [12, 19, 4], that is, in our case, a universal function $\theta : \mathbb{N} \to (\mathbb{N} \rightharpoonup \mathbb{N})$ such that:

$$\forall f : \mathbb{N} \rightharpoonup \mathbb{N}. \exists c. \forall xy. \theta cx \rhd y \;\leftrightarrow\; fx \rhd y$$

We can now show that the halting problem $H := \lambda c. \exists y. \theta cc \rhd y$ is undecidable. We can either assume θ to be abstract or to be an interpreter of a Turing-complete model of computation [11].

Strong G1. For Kleene's stronger result, consider the following two predicates:

$$A_1 := \{x \mid \theta xx \rhd 1\} \qquad A_0 := \{x \mid \theta xx \rhd 0\}$$

A_1 and A_0 are enumerable and recursively inseparable, that is, there is no decidable predicate D such that $A_1 \subseteq D$ and $A_0 \subseteq \overline{D}$. We assume that the formal system strongly separates A_1 and A_0, that is, there is a representation function $R : \mathbb{N} \to S$ such that:

$$x \in A_1 \;\to\; \vdash Rx \qquad x \in A_0 \;\to\; \vdash \neg Rx$$

Note that we do not need any form of soundness anymore. Consider a partial function that checks whether $\vdash Rx$ or $\vdash \neg Rx$ by enumerating all provable sentences and outputting 1 or 0 respectively. This function must diverge on some input c, because it would separate A_1 and A_0 otherwise, and therefore $\nvdash Rc$ and $\nvdash \neg Rc$. This input can be constructed explicitly using diagonalization and an application of CT. This would not be possible had we not assumed CT.

Instantiation. We use the same framework for first-order logic as in the instantiation of the folklore proof [7] with the theory of Robinson's Q [16]. We instantiate θ with an interpreter for μ-recursive functions, as described in [13]. Q weakly represents all predicates enumerable in μ (and by CT, all synthetically enumerable predicates) using Σ_1 formulas [13, 7].

Let φ_1, φ_0 be Σ_1-formulas that weakly represent A_1, A_0 respectively, that is $\forall c. c \in A_i \;\leftrightarrow\; Q \vdash \varphi_i(c)$. We can concretely assume that $\varphi_i(x) = \exists k. \psi_i(x, k)$, where ψ_i is Q-decidable, that is $\mathsf{Q} \vdash \psi_i(x, k) \lor \mathsf{Q} \vdash \neg\psi_i(x, k)$. We now apply Rosser's trick to φ_i, that is, we choose:

$$\Phi_i(x) := \exists k. \psi_i(x, k) \land \forall k' \leq k. \neg\psi_{1-i}(x, k')$$

Intuitively, Φ_i can be understood as "There is a proof k of $x \in A_i$, and there is no smaller proof of $x \in A_{1-i}$". Now, Φ_1 and Φ_0 both strongly separate A_1 and A_0:

$$x \in A_i \;\to\; \mathsf{Q} \vdash \Phi_i(x) \qquad x \in A_{1-i} \;\to\; \mathsf{Q} \vdash \neg\Phi_i(x)$$

Just as Rosser's trick allowed weakening the precondition of ω-consistency in Gödel's original proof of G1, it allows us to drop the requirement of soundness for the theory we are working with by relying on another form of representability. All properties required also hold for consistent extensions of Q, which allows us to show essential incompleteness of Q. It is also possible to obtain essential undecidability of Q by modifying the abstract results slightly.

This approach can be used to show incompleteness of other formal systems, such as CIC or other higher-order logics, as long as they weakly represent H and can apply Rosser's trick.

References

[1] Scott Aaronson. *Rosser's theorem via Turing machines*. Shtetl-Optimized. URL: https://scottaaronson.blog/?p=710 (visited on 28/02/2022).

[2] Andrej Bauer. "First Steps in Synthetic Computability Theory". In: *Electronic Notes in Theoretical Computer Science* 155 (2006), pp. 5–31.

[3] Thierry Coquand and Gérard Huet. "The calculus of constructions". In: *Information and Computation* 76.2 (1988), pp. 95–120.

[4] Yannick Forster. "Parametric Church's Thesis: Synthetic Computability Without Choice". In: *International Symposium on Logical Foundations of Computer Science*. 2022, pp. 70–89.

[5] Yannick Forster, Dominik Kirst and Gert Smolka. "On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2019, pp. 38–51.

[6] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für Mathematik und Physik* 38 (1931), pp. 173–198.

[7] Dominik Kirst and Marc Hermes. "Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq". In: *ITP 2021*. 2021.

[8] Stephen C. Kleene. "General recursive functions of natural numbers." In: *Mathematische Annalen* 112 (1936), pp. 727–742.

[9] Stephen C. Kleene. *Mathematical Logic*. Dover Publications, 1967.

[10] Stephen C. Kleene. "Recursive predicates and quantifiers". In: *Transactions of the American Mathematical Society* 53 (1943), pp. 41–73.

[11] Georg Kreisel. "Church's Thesis: A Kind of Reducibility Axiom for Constructive Mathematics". In: *Studies in Logic and the Foundations of Mathematics* 60 (1970), pp. 121–150.

[12] Georg Kreisel. "Mathematical Logic". In: *Journal of Symbolic Logic* 32.3 (1967), pp. 419–420.

[13] Dominique Larchey-Wendling and Yannick Forster. "Hilbert's Tenth Problem in Coq (Extended Version)". In: *Logical Methods in Computer Science* 18 (2022).

[14] Christine Paulin-Mohring. "Inductive definitions in the system Coq rules and properties". In: *Typed Lambda Calculi and Applications*. Springer, 1993, pp. 328–345.

[15] Fred Richman. "Church's Thesis Without Tears". In: *The Journal of Symbolic Logic* 48.3 (1983), pp. 797–803.

[16] Raphael Robinson. "An Essentially Undecidable Axiom System". In: *Proceedings of the International Congress of Mathematics*. 1950, pp. 729–730.

[17] Barkley Rosser. "Extensions of some theorems of Gödel and Church". In: *Journal of Symbolic Logic* 1.3 (1936), pp. 87–91.

[18] The Coq Development Team. *The Coq Proof Assistant*. Jan. 2022. DOI: 10.5281/zenodo.5846982.

[19] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics, Vol 1*. ISSN. Elsevier Science, 1988.

[20] Anatoly Vorobey. *First Incompleteness via computation: an explicit construction*. Foundations of Mathematics mailing list. URL: https://cs.nyu.edu/pipermail/fom/2021-September/022872.html (visited on 21/02/2022).

Synthetic Tait Computability for Simplicial Type Theory

Jonathan Weinberger[1], Benedikt Ahrens[2], Ulrik Buchholtz[3], and Paige North[4]

[1] Max Planck Institute for Mathematics, Bonn, Germany
weinberger@mpim-bonn.mpg.de
[2] Delft University of Technology, The Netherlands
b.p.ahrens@tudelft.nl
[3] University of Nottingham, UK
ulrik@ulrikbuchholtz.dk
[4] University of Pennsylvania, US
pnorth@upenn.edu

1 Introduction

Riehl and Shulman [13] introduced a simplicial extension of (homotopy) type theory to reason synthetically about $(\infty, 1)$-categories. Indeed, the semantics of this theory matches up with established results from homotopy theory, *i.e.*, the synthetic $(\infty, 1)$-categories in the theory correspond externally to internal $(\infty, 1)$-categories in an arbitrary given ∞-topos (implemented as complete Segal objects), cf. [10, 15, 23]. However, certain meta-theoretic properties of this simplicial type theory (STT), such as canonicity and normalization, have not been investigated yet.

In this work, we adapt the framework of *synthetic Tait computability (STC)* due to Sterling and Harper [18] to simplicial type theory, following the original work on cubical type theory (CTT) from Sterling–Angiuli [17] and Sterling [16]. The framework has previously been used to give syntax-invariant proofs of canonicity (cf. [14]) and normalization for cubical type theory, generalizing and internalizing previous accounts of normalization by evaluation (NBE) using the internal language of a sufficiently structured topos.

Contribution We explain how to define analogously a simplicial version of STC giving rise to a notion of computability topos for simplicial type theory. We'll also report on our progress in establishing a normalization proof à la [17, 16] for STT in this setting.

2 Simplicial Type Theory

Simplicial type theory (STT) due to [13] augments traditional Martin–Löf type theory by two features: (i) two additional pre-type layers (for directed *cubes* and (sub-)*shapes*), and (ii) *extension types*. The latter had been previously devised by Lumsdaine–Shulman in unpublished work. They can be understood as Π-*types with strict side conditions*: Given a shape inclusion $\Phi \hookrightarrow \Psi$, a type family $A : \Psi \to \mathcal{U}$ and a partial section $\sigma : \prod_{t:\Phi} A(t)$, the *extension type* $\langle \prod_\Psi A|^\Phi_\sigma \rangle$ can be understood as the type of all *totalizations of* σ up to judgmental equality, *i.e.*, all sections $\sigma' : \prod_{t:\Psi} A(t)$ such that $\sigma(t) \equiv \sigma'(t)$ for $t : \Phi$. This is also familiar from cubical type theory where path types are defined in a similar way. The rules of extension types are analogous to the ones for Π-types, but containing additional definitional equalities, cf. [13].

With this at hand, in STT one can then define a notion of weak composition of morphisms for a type A by requiring that the map $A^{\Delta^2} \to A^{\Lambda^2_1}$ induced by the inclusion of the shape $(\bullet \to \bullet \to \bullet)$ into the filled 2-simplex be a weak equivalence (*Segal condition*, cf. [11, 7]). This is a crucial ingredient for defining a synthetic notion of $(\infty, 1)$-category.

Building on fundamental parts of synthetic higher category theory developed in [13] there has been work on synthetic $(\infty, 1)$-categories in the simplicial setting [12, 4, 23, 9] and in the bicubical setting [22]. Kudasov is working on a prototype proof assistant supporting STT [8].

3 Simplicial Synthetic Tait Computability

Presentation as a fibered signature In his recent PhD thesis [16], Sterling develops a logical framework to define a variety of type theories. The idea is to present a type theory by a *signature*, which specifies abstractly the potential judgments to be formed. The actual admissible contexts of the type theory are organized into a *category of atomic contexts* which arises as a submodel of the syntactic model.

An abstract *signature* $\mathbb{S}$ is given by its collection of concrete *implementations* $U : \mathbb{S}$. The collection of signatures forms a category **SIG** whose morphisms are "functions" $U : \mathbb{S} \to \mathbb{T}$, $x \mapsto U(x)$. In fact, the signatures give rise to a type theory with dependent sums, products, identity types, and a terminal type $\mathbb{1}$. This implies that the category **SIG** is finitely complete.

Because of its multi-layered structure, STT is presented as a chain of projection maps from appropriate signatures:

$$\mathbb{STT} \xrightarrow{\pi_1} \mathbb{T}\mathfrak{opes} \xrightarrow{\pi_0} \mathbb{C}\mathfrak{ubes}$$

Simplicial STC and computability topos We adapt the axiomatization from [17] to the setting of simplicial rather than cubical type theory. One notable difference is that the simplicial interval $\mathbb{2}$ is *not* tiny, *i.e.*, exponentiation $(-)^{\mathbb{2}} : \mathcal{E} \to \mathcal{E}$ does not have a right adjoint.

Namely, we devise an axiomatization of an ambient category $\mathcal{E}$ whose internal language supports an appropriate notion of computability structure. This can be instantiated by a topos pushout (actually a gluing presheaf topos) of an open (syntactic) with a closed (semantic) subtopos, using an appropriate simplicial *figure shape*, as in [17, 16].

Towards normalization of STT Along the lines of [17, 16] we can then carry out a version of normalization by evaluation (NBE), based on an analogous notion of *stabilized neutrals* to capture the (more general) case of extension types.

In the classical picture of NBE, after Tait [21], one constructs for a type A a chain of maps (*reflection* and *reification*)

$$\mathrm{ne}(A) \to [\![A]\!] \to \mathrm{nf}(A)$$

from the *neutral terms* of type A to the (computational) semantics of terms of type A, and from there to the *normal forms* of type A (cf. also [5, 2, 20, 6, 1, 3, 19]).

For CTT, it was the insight of Sterling–Angiuli that the conditions for a term to be neutral could not be formulated reasonably. However, they were able to instead capture the conditions for a term to *cease* to be neutral. Informally, if p denotes a path term and i an interval variable, the neutral path application term $p(i)$ would cease to be neutral in case $i \equiv 0$ or $i \equiv 1$ (since this would force another computation). This gives rise to a modified version of Tait's method, called *stabilized* Tait yoga.

We are re-using this idea for the case of STT where this *frontier of instability* for application $\tau(t)$ of a section $\tau : \langle \prod_\Psi A|^\Phi_\sigma \rangle$ to a tope variable $t : \Psi$ depends on t satisfing the condition $\varphi(t)$ defining the distinguished subshape Φ.

With those modifications, our current progress indicates that the methods by Sterling–Angiuli and Sterling to prove normalization carry over well to the simplicial setting.[1] We will report on our progress in this program.

References

[1] Andreas Abel. *Normalization by evaluation: Dependent types and impredicativity*. Habilitation, Ludwig-Maximilians-Universität München, 2013. https://www.cse.chalmers.se/~abela/habil.pdf.

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

[3] Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for type theory, in type theory. *Log. Methods Comput. Sci.*, 13(4):26, 2017. Id/No 1.

[4] Ulrik Buchholtz and Jonathan Weinberger. Synthetic fibered $(\infty, 1)$-category theory, 2021. https://arxiv.org/abs/2105.01724.

[5] Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Math. Struct. Comput. Sci.*, 8(2):153–192, 1998.

[6] Marcelo Fiore. 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*, pages 26–37, 2002.

[7] André Joyal and Myles Tierney. Quasi-categories vs Segal spaces. *Contemporary Mathematics*, 431(277-326):10, 2007.

[8] Nikolai Kudasov. `rzk`. 2021. Prototype interactive proof assistant based on a type theory for synthetic ∞-categories. https://github.com/fizruk/rzk.

[9] César Bardomiano Martínez. Limits and colimits of synthetic ∞-categories, 2022. https://arxiv.org/abs/2202.12386.

[10] Nima Rasekh. Complete Segal objects. 2018. https://arxiv.org/abs/1805.03561.

[11] Charles Rezk. A model for the homotopy theory of homotopy theory. *Transactions of the American Mathematical Society*, 353(3):973–1007, 2001.

[12] Emily Riehl, Evan Cavallo, and Christian Sattler. On the directed univalence axiom, 2018. Talk at the AMS Special Session on Homotopy Type Theory, Joint Mathematics Meetings. https://emilyriehl.github.io/files/jmm2018.pdf.

[13] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories. *Higher Structures*, 1(1):147–224, May 2017.

[14] Christian Sattler, Simon Huber, and Thierry Coquand. Canonicity and homotopy canonicity for cubical type theory. *Logical Methods in Computer Science*, 18:28:1–28:35, 2022.

[15] Michael Shulman. All $(\infty, 1)$-toposes have strict univalent universes, 2019. https://arxiv.org/abs/1904.07004.

[16] Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University, 2021. CMU technical report CMU-CS-21-142, https://doi.org/10.5281/zenodo.5709837.

[17] Jonathan Sterling and Carlo Angiuli. Normalization for cubical type theory. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–15. IEEE, 2021.

[1] Note that, in contrast to [13], we do not assume any axioms in the present version of simplicial type theory such as function extensionality (neither for ordinary function nor extension types), let alone univalence, as these would obstruct canonicity.

[18] Jonathan Sterling and Robert Harper. Logical relations as types: Proof-relevant parametricity for program modules. *J. ACM*, 68(6):41:1–41:47, 2021.

[19] Jonathan Sterling and Bas Spitters. Normalization by gluing for free λ-theories. 2018. https://arxiv.org/abs/1809.08646.

[20] Thomas Streicher. Categorical intuitions underlying semantic normalisation proofs. In Olivier Danvy and Peter Dybjer, editors, *1998 APPSEM Workshop on Normalization by Evaluation NBE '98*, pages 9–10. Department of Computer Science, Aarhus University, 1998.

[21] W. W. Tait. Intensional interpretations of functionals of finite type. I. *J. Symb. Log.*, 32:198–212, 1967.

[22] Matthew Z. Weaver and Daniel R. Licata. A constructive model of directed univalence in bicubical sets. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, pages 915–928, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Jonathan Weinberger. *A Synthetic Perspective on* $(\infty, 1)$*-Category Theory: Fibrational and Semantic Aspects*. PhD thesis, Technische Universität, Darmstadt, 2022. https://doi.org/10.26083/tuprints-00020716.

Synthetic Turing Reducibility in CIC

Yannick Forster[1,2] and Dominik Kirst[1]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] Inria, Gallinette Project-Team, Nantes, France

Abstract

We discuss a definition of Turing reducibility in synthetic computability carried out in CIC, the type theory underlying Coq. CIC distinguishes between functions (which can be assumed to all be computable) and total functional relations, allowing for a definition of Turing reducibility via continuous Turing functionals, based on an idea of Bauer.

Turing reductions A problem P is Turing-reducible to a problem Q if P can be solved by a machine which has access to an oracle for Q (the notion was introduced in Turing's PhD thesis [21], but popularised by Post [16]). In less operational models of computation like μ-recursive functions, Turing reductions can be described via functionals $F\colon(\mathbb{N}\rightharpoonup\mathbb{N})\to(\mathbb{N}\rightharpoonup\mathbb{N})$ where the value $F(\alpha)x$ is obtained from the potentially non-computable α and the natural number x solely by the usual computable operations of μ-recursive functions. As a consequence, a μ-recursive functional sends a μ-recursive function α to a μ-recursive function $F(\alpha)$. Kleene [12] and Davis [5] both proved that any μ-recursive functional F is continuous: Whenever $F(\alpha)x\triangleright_\mu y$, then there is a list L included in the domain of α such that whenever β returns the same values on L as *alpha*, then also $F(\beta)x\triangleright_\mu y$. Formally:

$$F(\alpha)x\triangleright_\mu y\to\exists L\colon\mathbb{L}\mathbb{N}.\ (\forall x\in L.\ \exists y.\ \alpha x\triangleright y)\wedge\forall\beta.\ (\forall x\in L.\ \alpha x=\beta x)\to F(\beta)x\triangleright_\mu y$$

To the best of our knowledge Turing reducibility and (continuous) μ-recursive functionals have not been studied in type theory or constructive (reverse) mathematics, or formalised using machine-checked proofs in a proof assistant. The main hinderance regarding computability theory is the ubiquitous use of the (informal) Church-Turing thesis, which connects "intuitive calculability" with a formal notion (computability in a defined model of computation).

Synthetic computability Synthetic mathematics in general offers a solution for situations where analytic encodings of notions muddle the clear view at the essence of concepts. Introduced by Richman [18] and popularised by Richman, Bridges, and Bauer [4, 1, 2, 3] in synthetic computability one assumes an axiom amounting to imposing a universal function for the space $\mathbb{N}\to\mathbb{N}$, or, equivalently, for $\mathbb{N}\rightharpoonup\mathbb{N}$. For instance, one can state the axiom EPF as

$$\exists\theta\colon\mathbb{N}\to(\mathbb{N}\rightharpoonup\mathbb{N}).\ \forall f\colon\mathbb{N}\rightharpoonup\mathbb{N}.\ \exists c.\ \theta_c\equiv f$$

for a suitable notion of partial functions (e.g. where partial values over X are modelled by step-indexing as monotonous sequences from $\mathbb{N}$ to the option type over X). EPF is closely related to the well-studied axiom CT ("Church's thesis" [14, 20]) in constructive mathematics, which postulates that *all* functions are μ-recursive. CT immediately implies EPF, because θ can be taken to be the universal function for μ-recursive functions, see [8].

In synthetic computability, one can define an undecidable but enumerable problem $\mathcal{K}$, where both enumerability and undecidability are formalised solely in terms of functions. Results like Rice's theorem [1, 8] and Myhill's isomorphism theorem [9], are relatively easy to prove in synthetic computability theory. Synthetic definitions of many-one and truth-table reducibility can be given by just dropping "computable" from textbook definitions, and we have constructed synthetic solutions for Post's problem for many-one and truth-table reducibility [9]. Turing reducibility is less easily synthesised, due to the notion of an oracle. For instance, under the presence of EPF, a functional $F\colon(\mathbb{N}\rightharpoonup\mathbb{N})\to(\mathbb{N}\rightharpoonup\mathbb{N})$ only acts on computable input, thus it cannot be seen as a Turing reduction acting on a (certainly non-computable) oracle input.

Synthetic Turing reducibility We use the intuition behind the mentioned Kleene/Davis theorem to define synthetic Turing reducibility, based on *continuous Turing functionals*. We follow an idea by Bauer [3] and define Turing functionals based on a two-layered approach: They consist of a (continuous) functional $F\colon (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$ mapping functional relations $Y \rightsquigarrow \mathbb{B}$ to functional relations $X \rightsquigarrow \mathbb{B}$ factoring through a (then also continuous) computational core $F'\colon (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B})$ (see also [7] for a more detailed treatment).

Formally a Turing functional $F\colon (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$ ...

1. ... is *continuous* if: $\forall R\colon Y \rightsquigarrow \mathbb{B}.\forall x\colon X.\forall b\colon \mathbb{B}.\ FRxb \to \exists L : \mathbb{L}Y.\ (\forall y \in L.\exists b.\ Ryb) \wedge \forall R'\colon Y \rightsquigarrow \mathbb{B}.\ (\forall y \in L.\forall b.\ Ryb \to R'yb) \to FR'xb$
2. ... factors through a *computational core* $F'\colon (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B})$ if:

$$\forall f\colon Y \rightharpoonup \mathbb{B}.\forall R\colon Y \rightsquigarrow \mathbb{B}.\ f \textit{ computes } R \to F'f \textit{ computes } FR$$

where $f\colon Z_1 \rightharpoonup Z_2$ *computes* a functional relation $R\colon Z_1 \rightsquigarrow Z_2$ if $\forall xy.\ Rxy \leftrightarrow fx \triangleright y$.

Note that we follow the same intuition as for μ-recursive functionals to define continuity here. A synthetic Turing reduction from a predicate $p\colon X \to \mathbb{P}$ to a predicate $q\colon Y \to \mathbb{P}$ maps the characteristic relation of q ($\lambda xb.\ qx \leftrightarrow b = \mathsf{true}$) to the characteristic relation of p.

The definition makes crucial use of the fact that functional relations are completely distinct from functions since unique choice is not provable in CIC.

Our definition of Turing reducibility is work-in-progress: We were able to prove various results regarding Turing reducibility with machine-checked proofs in Coq, see below.

Bauer's definition of Turing reducibility Our definition is based on (computable) functional relations $X \rightsquigarrow \mathbb{B}$, whereas Bauer's definition [3] is based on disjoint pairs of (enumerable) predicates $X \to \mathbb{P}$ and using an order-theoretic definition of continuity. Proving our definition of Turing reducibility equivalent without considering continuity is straightforward [7, §9.6]. We also have an equivalence proof w.r.t. continuity now for a variant of Bauer's definition of continuity, following well-known ideas explained for instance in Rogers' book [19, II.3.2].

Machine-checked results Turing reducibility is reflexive, transitive, and transports undecidability. Every truth-table reduction gives rise to a Turing reduction. When a Turing reduction is total for all total inputs and the L in compactness is computable, it is in fact a truth-table reduction (a result Rogers attributes to Nerode [19, Thm. XIX]). Lastly, the hypersimple deficiency predicate of $\mathcal{K}$ is Turing reduction complete, showing that Turing reducibility is strictly more general than truth-table reducibility. All results are explained in [7, §10].

Open questions However, at least three more results for Turing reducibility are needed to have confidence that our synthetic rendering is correct. The Kleene-Post theorem [13], stating that there are incomparable Turing-reducibility degrees, Post's theorem [17], connecting Turing reducibility via the Turing jump to the arithmetical hierarchy, and the Friedberg-Muchnik theorem [10, 15], settling Post's problem by proving that there exists an enumerable, undecidable, but Turing-reducibility incomplete predicate. Central for all three results is, in contrast to the results we have already proved, an enumerator of cores of Turing functional.

Towards a universal core We require a function $\zeta\colon \mathbb{N} \to ((\mathbb{N} \rightharpoonup \mathbb{N}) \to \mathbb{N} \rightharpoonup \mathbb{N})$ which enumerates at least all possible cores of Turing functionals. Certain variants of such a function are known to be impossible: A computable modulus of continuity applying to itself would be inconsistent in type theory [6]. We are attacking the problem from two directions: First, we are trying to construct ζ from θ either directly or by enriching the definition of Turing reductions with more computational requirements for cores, e.g. by asking for a computable modulus of continuity for F', which possibly has to be continuous itself. Secondly, in a concurrently submitted abstract we are describing our work-in-progress proofs of synthetic variants of the Kleene-Post and Post's theorem, identifying sufficient properties of a universal function ζ [11].

References

[1] Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006. doi:10.1016/j.entcs.2005.11.049.

[2] Andrej Bauer. On fixed-point theorems in synthetic computability. *Tbilisi Mathematical Journal*, 10(3):167–181, 2017. doi:10.1515/tmj-2017-0107.

[3] Andrej Bauer. Synthetic mathematics with an excursion into computability theory (slide set). *University of Wisconsin Logic seminar*, 2020. URL: http://math.andrej.com/asset/data/madison-synthetic-computability-talk.pdf.

[4] Douglas Bridges and Fred Richman. *Varieties of constructive mathematics*, volume 97. Cambridge University Press, 1987. doi:10.1017/CBO9780511565663.

[5] Martin D. Davis. *Computability and Unsolvability*. McGraw-Hill Series in Information Processing and Computers. McGraw-Hill, 1958.

[6] Martín Hötzel Escardó. Computability of continuous solutions of higher-type equations. In Klaus Ambos-Spies, Benedikt Löwe, and Wolfgang Merkle, editors, *Mathematical Theory and Computational Practice, 5th Conference on Computability in Europe, CiE 2009, Heidelberg, Germany, July 19-24, 2009. Proceedings*, volume 5635 of *Lecture Notes in Computer Science*, pages 188–197. Springer, 2009. doi:10.1007/978-3-642-03073-4_20.

[7] Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021. URL: https://ps.uni-saarland.de/~forster/thesis.

[8] Yannick Forster. Parametric Church's Thesis: Synthetic computability without choice. In *International Symposium on Logical Foundations of Computer Science*, pages 70–89. Springer, 2022. doi:10.1007/978-3-030-93100-1_6.

[9] Yannick Forster, Felix Jahn, and Gert Smolka. A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq (Full Version). preprint, February 2022. URL: https://hal.inria.fr/hal-03580081.

[10] Richard M Friedberg and Hartley Rogers Jr. Reducibility and completeness for sets of integers. *Mathematical Logic Quarterly*, 5(7-13):117–125, 1959. doi:10.1002/malq.19590050703.

[11] Dominik Kirst, Niklas Mück, and Yannick Forster. Synthetic versions of the Kleene-Post and Post's theorem. In *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, 2022.

[12] Stephen C. Kleene. *Introduction to metamathematics*, volume 483. van Nostrand New York, 1952.

[13] Steven C. Kleene and Emil L. Post. The upper semi-lattice of degrees of recursive unsolvability. *The Annals of Mathematics*, 59(3):379, May 1954. doi:10.2307/1969708.

[14] Georg Kreisel. Mathematical logic. *Lectures in modern mathematics*, 3:95–195, 1965. doi:10.2307/2315573.

[15] Albert Abramovich Muchnik. On strong and weak reducibility of algorithmic problems. *Sibirskii Matematicheskii Zhurnal*, 4(6):1328–1341, 1963.

[16] Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *bulletin of the American Mathematical Society*, 50(5):284–316, 1944. doi:10.1090/S0002-9904-1944-08111-1.

[17] Emil L. Post. Degrees of recursive unsolvability - preliminary report. In *Bulletin of the American Mathematical Society*, volume 54, pages 641–642. American Mathematical Society (AMS), 1948.

[18] Fred Richman. Church's thesis without tears. *The Journal of symbolic logic*, 48(3):797–803, 1983. doi:10.2307/2273473.

[19] Hartley Rogers. Theory of recursive functions and effective computability. 1987.

[20] Anne Sjerp Troelstra and Dirk van Dalen. Constructivism in mathematics. vol. i. *Studies in Logic and the Foundations of Mathematics*, 26, 1988.

[21] Alan Mathison Turing. Systems of logic based on ordinals. *Proceedings of the London mathematical society*, 2(1):161–228, 1939. doi:10.1112/plms/s2-45.1.161.

Synthetic Versions of the Kleene-Post and Post's Theorem

Dominik Kirst[1], Niklas Mück[1], and Yannick Forster[1,2]

[1] Saarland University, Saarland Informatics Campus
[2] Inria, Gallinette Project-Team, Nantes, France

Abstract

We discuss our ongoing formalisation of the Kleene-Post theorem ([6], establishing incomparable Turing degrees) and Post's theorem ([9], connecting the arithmetic hierarchy with Turing degrees) using synthetic computability theory in constructive type theory.

Synthetic Oracle Machines We briefly outline the synthetic rendering of oracle machines as described in a related abstract [4], adjusting [2], based on a similar proposal by Bauer [1]. The main idea is that an oracle machine R can be represented as a function operating on functional relations $A : \mathbb{N} \rightsquigarrow \mathbb{B}$ relating (some) natural numbers $n : \mathbb{N}$ to (unique) boolean values $b : \mathbb{B}$:

$$R : (\mathbb{N} \rightsquigarrow \mathbb{B}) \to \mathbb{N} \rightsquigarrow \mathbb{B}$$

The input relation acts as oracle that can be accessed to describe the returned relation. To ensure that this description is effective, we require R to return computable output for computable input, captured as partial functions $f : \mathbb{N} \rightharpoonup \mathbb{B}$, by imposing a computational core

$$r : (\mathbb{N} \rightharpoonup \mathbb{B}) \to \mathbb{N} \rightharpoonup \mathbb{B} \qquad \text{with} \qquad \forall f.\, R\, f = r\, f.$$

Note that here and in the remainder of this text we freely identify partial functions $f : \mathbb{N} \rightharpoonup \mathbb{B}$ with their (functional) graphs $\lambda n b.\, f n = b$, reusing the equality symbol for evaluation of f. To further rule out exotic behaviour, we require R to be continuous in the following sense:

$$\forall (A : \mathbb{N} \rightsquigarrow \mathbb{B}).\forall (n : \mathbb{N}).\forall (b : \mathbb{B}).\ R\, A\, n\, b \to \exists (L : \mathbb{N}^*).\, L \subseteq \mathsf{dom}(A) \wedge \forall A'.\, A' =_L A \to R\, A'\, n\, b$$

Continuity in this sense expresses that from any terminating run $R\, A\, n\, b$ one can extract a list L of queries to which the oracle A replied, such that $R\, A'\, n$ terminates for all oracles A' agreeing with A on L with the same value b. Observed externally, by all these restrictions and according to the classical, syntactic definition we narrow down the amount of oracle machines to countable extent. In fact, we make this limitation available internally by assuming an enumeration r_n of all computational cores. We currently investigate for which formulations a variant of Church's thesis [7, 10, 11, 3] is enough to obtain such an enumerator.

Given $A, B : \mathbb{N} \to \mathbb{P}$, we call R a Turing reduction from A to B if $R\, B = A$ (reinterpreting predicates as functional relations) and write $A \preceq_T B$ if a Turing reduction from A to B exists. We assume extensionality of functions and relations.

Kleene-Post Theorem To establish incomparable Turing degrees, we adapt the proof given in Odifreddi's textbook [8] to our synthetic setting. The usual strategy is to obtain them as the unions $A := \bigcup_{n:\mathbb{N}} \sigma_n$ and $B := \bigcup_{n:\mathbb{N}} \tau_n$ of cumulative increasing sequences σ_n and τ_n of boolean strings such that the former take care that no r_n induces a reduction $B \preceq_T A$ and the latter conversely rule out $A \preceq_T B$. Naturally, in our synthetic setting we are not able to define these sequences as computable functions $\mathbb{N} \to \mathbb{B}^*$, as this would force A and B decidable. Instead, we characterise both sequences simultaneously with an inductive predicate $\triangleright : \mathbb{N} \to \mathbb{B}^* \to \mathbb{B}^* \to \mathbb{P}$ such that $n \triangleright (\sigma, \tau)$ represents σ_n as σ and τ_n as τ, by adding to $0 \triangleright (\epsilon, \epsilon)$ the rules:

$$\frac{2n \triangleright (\sigma, \tau) \quad \sigma' \geq \sigma \quad b = r_n\, \sigma'\, |\tau|}{2n+1 \triangleright (\sigma', \tau +\!\!+ [\neg b])}\text{E1} \qquad \frac{2n \triangleright (\sigma, \tau) \quad \neg(\exists \sigma' b.\, \sigma' \geq \sigma \wedge b = r_n\, \sigma'\, |\tau|)}{2n+1 \triangleright (\sigma, \tau +\!\!+ [0])}\text{E2}$$

$$\frac{2n+1 \triangleright (\sigma, \tau) \quad \tau' \geq \tau \quad b = r_n\, \tau'\, |\sigma|}{2n+2 \triangleright (\sigma +\!\!+ [\neg b], \tau')}\text{O1} \qquad \frac{2n+1 \triangleright (\sigma, \tau) \quad \neg(\exists \tau' b.\, \tau' \geq \tau \wedge b = r_n\, \tau'\, |\sigma|)}{2n+2 \triangleright (\sigma +\!\!+ [0], \tau)}\text{O2}$$

In every even step with $2n \triangleright (\sigma, \tau)$ the sequences are extended such that r_n applied to any prefix of A differs from any prefix of B at position $|\tau|$, either by flipping the result if r_n already converges on some extension $\sigma' \geq \sigma$ (E1) or by setting a dummy value if r_n diverges on all extensions (E2). Dually, in every odd step with $2n+1 \triangleright (\sigma', \tau')$ it is taken care that r_e applied to any prefix of B differs from any prefix of A.

We state the central lemma used to show $B \not\preceq_T A$, a dual version yields $A \not\preceq_T B$.

Lemma 1. *Let R be an oracle machine factoring through the computational core r_n. If further given $2n \triangleright (\sigma, \tau)$ and $2n+1 \triangleright (\sigma', \tau')$, then $B\,|\tau|\,b$ implies $\neg(R\,A\,|\tau|\,b)$.*

Theorem 1 (Kleene-Post)**.** *There are predicates A, B such that neither $A \preceq_T B$ nor $B \preceq_T A$.*

Proof. Suppose that $B \preceq_T A$, so $R\,A = B$ for some oracle machine R with core r_n. Given that we try to derive a contradiction, we can argue classically enough to obtain $2n \triangleright (\sigma, \tau)$, $2n+1 \triangleright (\sigma', \tau')$, and $B\,|\tau|\,b$. Then by Lemma 1 we obtain $\neg(R\,A\,|\tau|\,b)$, contradicting $R\,A = B$. □

Post's Theorem To connect the arithmetical hierarchy with the structure of Turing degrees, we again follow a usual textbook presentation translated to constructive type theory. To be able to state the theorem in our setting, we render all involved notions synthetically.

First, we represent the arithmetical hierarchy with a mutually inductive predicate:

$$\frac{f : \mathbb{N}^k \to \mathbb{B}}{\Sigma_0^k(\lambda \vec{n}.\, f\,\vec{n} = \mathsf{true})} \qquad \frac{f : \mathbb{N}^k \to \mathbb{B}}{\Pi_0^k(\lambda \vec{n}.\, f\,\vec{n} = \mathsf{true})} \qquad \frac{\Pi_n^{k+1} p}{\Sigma_{n+1}^k(\lambda \vec{n}.\, \exists x.\, p\,(x :: \vec{n}))} \qquad \frac{\Sigma_n^{k+1} p}{\Pi_{n+1}^k(\lambda \vec{n}.\, \forall x.\, p\,(x :: \vec{n}))}$$

The first two rules assert that k-ary decidable predicates form the base of the hierarchy. The third rule states that for a Π_n predicate p of arity $k+1$ the k-ary predicate obtained by capturing the first variable of p by an existential quantifier is Σ_{n+1}. The fourth rule dually expresses how a Σ_n predicate is turned into Π_{n+1} with a universal quantifier. As a sanity check, using a form of Church's thesis for a concrete model of computation, we can show the equivalence of our synthetic characterisation of the arithmetic hierarchy with a more conventional definition using first-order fomulas in the language of arithmetic, as mechanised in [5].

Secondly, we define the Turing jump A' of a predicate A using the core enumeration r_n:

$$A' := \lambda n.\, \exists R.\, (\forall f.\, R\,f = r_n\,f) \wedge R\,A\,n\,\mathsf{true}$$

This definition expresses the self-halting problem for oracle machines as it contains exactly those numbers n such that the n-th oracle machine R (as characterised by r_n) used with an oracle for A accepts n. We denote the n-th Turing jump of the empty predicate by $\emptyset^{(n)}$.

Finally, we say that A is semi-decidable relative to B if there is an oracle machine R with

$$\forall n.\, A\,n \leftrightarrow R\,B\,n\,\mathsf{true}.$$

The hardest part of Post's theorem is to show that $R\,A$ is Σ_1 relative to A by showing:

Lemma 2. *Given an oracle machine R with core r, termination $R\,A\,n\,b$ is equivalent to*

$$\exists L_{\mathsf{true}} L_{\mathsf{false}}.\, (\forall n \in L_{\mathsf{true}}.\, A\,b\,\mathsf{true}) \wedge (\forall n \in L_{\mathsf{false}}.\, A\,b\,\mathsf{false}) \wedge r\,(\mathsf{lookup}\,L_{\mathsf{true}}\,L_{\mathsf{false}})\,n = b$$

where $\mathsf{lookup}\,L_{\mathsf{true}}\,L_{\mathsf{false}}\,n$ *returns* true *if* $n \in L_{\mathsf{true}}$, false *if* $n \in L_{\mathsf{false}}$, *and diverges otherwise.*

We conclude Post's theorem in a common formulation, employing our synthetic definitions.

Theorem 2 (Post)**.** *Assuming LEM ($\forall p.\, p \vee \neg p$), the following can be shown:*

- *A unary predicate A is Σ_{n+1} iff it is semi-decidable relative to $\emptyset^{(n)}$.*
- *If A is Σ_n, then $A \preceq_T \emptyset^{(n)}$. If $n > 0$ already $A \preceq_m \emptyset^{(n)}$ for synthetic many-one reductions.*

In our current mechanisation, we assume LEM to allow switching between Σ_n and Π_n by complementation. We currently investigate how this assumption can be weakened or eliminated.

References

[1] Andrej Bauer. Synthetic mathematics with an excursion into computability theory. University of Wisconsin Logic seminar, 2021. URL: http://math.andrej.com/asset/data/madison-synthetic-computability-talk.pdf.

[2] Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021. URL: https://ps.uni-saarland.de/~forster/thesis.

[3] Yannick Forster. Parametric Church's Thesis: Synthetic computability without choice. In *International Symposium on Logical Foundations of Computer Science*, pages 70–89. Springer, 2022. doi:10.1007/978-3-030-93100-1_6.

[4] Yannick Forster and Dominik Kirst. Synthetic Turing reducibility in constructive type theory. 28th International Conference on Types for Proofs and Programs (TYPES 2022), 2022.

[5] Dominik Kirst and Marc Hermes. Synthetic undecidability and incompleteness of first-order axiom systems in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.23.

[6] S. C. Kleene and Emil L. Post. The upper semi-lattice of degrees of recursive unsolvability. *The Annals of Mathematics*, 59(3):379, May 1954. doi:10.2307/1969708.

[7] Georg Kreisel. Mathematical logic. *Lectures in modern mathematics*, 3:95–195, 1965. doi:10.2307/2315573.

[8] Piergiorgio Odifreddi. *Classical recursion theory: The theory of functions and sets of natural numbers*. Elsevier, 1992.

[9] Emil L. Post. Degrees of recursive unsolvability - preliminary report. In *Bulletin of the American Mathematical Society*, volume 54, pages 641–642. American Mathematical Society (AMS), 1948.

[10] Fred Richman. Church's thesis without tears. *The Journal of symbolic logic*, 48(3):797–803, 1983.

[11] Anne Sjerp Troelstra and Dirk van Dalen. Constructivism in mathematics. vol. i. *Studies in Logic and the Foundations of Mathematics*, 26, 1988.

The 4th Homotopy Group of the 3-Sphere in Cubical Agda

Axel Ljungström and Anders Mörtberg

Stockholm University, Stockholm, Sweden
{axel.ljungstrom, anders.mortberg}@math.su.se

One of the most extensive works on synthetic homotopy theory in HoTT/UF so far can be found in the PhD thesis of Brunerie [1]. The main result of the thesis is that $\pi_4(\mathbb{S}^3)$, the fourth homotopy group of the 3-sphere, is isomorphic to $\mathbb{Z}/2\mathbb{Z}$. For many years, this result has remained unformalised. The two main problems seem to have been:

1. Some theorems/constructions have simply been very hard to formalise, despite having relatively detailed proofs in Brunerie's thesis.
2. Many results in the latter half of the thesis concerning cohomology rely on theorems concerning the so called *smash product (of two types)*. In particular, they rely on the associativity of the smash product – a result whose proof is mostly omitted in Brunerie's thesis.

In this talk, we will present a full formalisation[1] of Brunerie's theorem carried out in Cubical Agda [5]. Our solutions to the aforementioned problems can be summarised as follows.

1. Streamline some of the proofs – in particular one, concerning an application of the so called James construction.
2. Use the cohomology theory presented in [2] in order to escape the smash product induced coherence hell.

We will present our formalisation with an emphasis on the mathematical changes to Brunerie's proof. To this end, we will give a brief overview of the main ideas of Brunerie's original proof and pay special attention to those details which have been simplified in our work. The proof is divided into two parts.

Part 1: Constructing the Brunerie number. In the first part, Brunerie constructs what is now known as the *Brunerie number*, an integer $\beta : \mathbb{Z}$ such that $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/\beta\mathbb{Z}$. To this end, he defines two maps

$$W : \mathbb{S}^3 \to \mathbb{S}^2 \vee \mathbb{S}^2 \qquad\qquad \nabla : \mathbb{S}^2 \vee \mathbb{S}^2 \to \mathbb{S}^2$$

where $\vee$ denotes the wedge sum [4, Section 6.8]. The first half of Brunerie's thesis is concerned with constructing a chain of isomorphisms $\pi_4(\mathbb{S}^3) \cong \pi_3(J_2(\mathbb{S}^2)) \cong \pi_3(\mathsf{cofib}(\nabla \circ W))$, where $J_2(\mathbb{S}^2)$ and $\mathsf{cofib}(\nabla \circ W)$ respectively are the pushouts of the spans $\mathbb{S}^2 \times \mathbb{S}^2 \hookleftarrow \mathbb{S}^2 \vee \mathbb{S}^2 \to \mathbb{S}^2$ and $1 \longleftarrow \mathbb{S}^3 \xrightarrow{\nabla \circ W} \mathbb{S}^2$. Brunerie then applies the *Blakers-Massey theorem* [3] and the long exact sequence of homotopy groups [4, Theorem 8.4.6] to obtain an exact sequence

$$\underbrace{\pi_3(\mathbb{S}^3)}_{\cong \langle e \rangle} \xrightarrow{\pi_3(\nabla \circ W)} \underbrace{\pi_3(\mathbb{S}^2)}_{\cong \langle h \rangle} \longrightarrow \pi_3(\mathsf{cofib}(\nabla \circ W)) \longrightarrow 1$$

where $e : \pi_3(\mathbb{S}^3)$ and $h : \pi_3(\mathbb{S}^2)$ are generators of the homotopy groups. Using the above sequence, the Brunerie number may be defined as follows.

[1] See https://github.com/agda/cubical/blob/master/Cubical/Homotopy/Group/Pi4S3/Summary.agda

Definition/Theorem 1. *Define* $\eta := \pi_3(\nabla \circ W)(e) : \pi_3(\mathbb{S}^2)$. *The Brunerie number* β *is the unique integer satisfying* $\eta = \beta \cdot h$. *In particular, it satisfies* $\pi_3(\mathsf{cofib}(\nabla \circ W)) \cong \pi_4(\mathbb{S}^3) \cong \mathbb{Z}/\beta\mathbb{Z}$

We have formalised the above result in Cubical Agda. In the formalisation, we deviate in one major way. The crucial step in the above proof is the isomorphism $\pi_4(\mathbb{S}^3) \cong \pi_3(J_2(\mathbb{S}^2))$. In Brunerie's thesis, this result uses the *James construction* [1, Chapter 3]. In our formalisation, however, we have been able to avoid the general James construction altogether. Instead, we construct the isomorphism directly, via the following lemma.

Lemma 2. *There is a (non-trivial) family of equivalences* $F : \mathbb{S}^2 \to \left\|J_2(\mathbb{S}^2)\right\|_3 \simeq \left\|J_2(\mathbb{S}^2)\right\|_3$ *such that* $F(*_{\mathbb{S}^2})$ *is the identity equivalence.*

Theorem 3. $\Omega \left\|\mathbb{S}^3\right\|_4 \simeq \left\|J_2(\mathbb{S}^2)\right\|_3$. *Hence, we also have* $\pi_4(\mathbb{S}^3) \cong \pi_3(J_2(\mathbb{S}^2))$.

Proof. Via Lemma 2 and the *encode-decode* method [4, Section 8.9]. □

The above theorem allows us to skip most of pages 67-81 in Brunerie's thesis and were relatively straightforward to formalise in Cubical Agda. The price we pay is, of course, that the theory we develop is less general.

Part 2: Proving that $|\beta| = 2$. For this step, we need to show for an isomorphism of our choice $\psi : \pi_3(\mathbb{S}^2) \cong \mathbb{Z}$ that $|\psi(\eta)| = 2$. There is a canonical one, coming from the long exact sequence of homotopy groups, but its action of η is rather unclear. Instead, Brunerie uses the so called *Hopf Invariant*, $\mathsf{HI} : \pi_3(\mathbb{S}^2) \to \mathbb{Z}$. He then proves the following three facts.

Theorem 4. *The Hopf invariant is a group homomorphism.*

Theorem 5. *For the generator* $h : \pi_3(\mathbb{S}^2)$, *we have* $|\mathsf{HI}(h)| = 1$. *Hence, the Hopf invariant is an isomorphism.*

Theorem 6. *We have* $\mathsf{HI}(\eta) = \pm 2$, *and hence* $|\beta| = 2$.

Theorem 4–6 all rely on various results concerning the cup product $\smile : H^n(A) \to H^m(A) \to H^{n+m}(A)$, where $H^n(A)$ is the *nth integral cohomology group of a type* A (see e.g. [2, Chapter 3]). In particular, they rely on the following result.

Theorem 7. *The cup product is associative, distributive and graded-commutative.*

In Brunerie's thesis, this theorem is stated, but its proof relies partly on an omitted proof of the associativity of the smash product. Theorem 7 *was*, however, proved and formalised by the authors and Brunerie in [2, Section 4.2], using an alternative definition of the cup product. This has allowed us to continue and complete the formalisation of Brunerie's original proof. This includes, apart from Theorems 4–6, constructions like the (iterated) Hopf construction and the Gysin sequence [1, Chapter 6]. On the way, we found some new "baby Brunerie numbers" which, like the Brunerie number, are integers definable in Cubical Agda, but which we are not able to normalise.

As is often the case, the formalisation caught some minor typos is Brunerie's thesis (e.g. the definition of $W_{A,B}$ on page 82 is ill-typed in the `push` case). However, on the whole, we found the proofs to be correct and "formalisation ready".

References

[1] Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory.* PhD thesis, Université Nice Sophia Antipolis, 2016.

[2] Guillaume Brunerie, Axel Ljungström, and Anders Mörtberg. Synthetic integral cohomology in cubical agda. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[3] Kuen-Bang Hou (Favonia), Eric Finster, Daniel R. Licata, and Peter LeFanu Lumsdaine. A Mechanization of the Blakers-Massey Connectivity Theorem in Homotopy Type Theory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 565–574, New York, NY, USA, 2016. ACM.

[4] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Self-published, Institute for Advanced Study, 2013.

[5] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):87:1–87:29, August 2019.

The Münchhausen method and combinatory type theory

Thorsten Altenkirch[1], Ambrus Kaposi[2]*, Artjoms Šinkarovs[3]†, and Tamás Végh[2]

[1] School of Computer Science, University of Nottingham, UK
psztxa@nottingham.ac.uk
[2] Eötvös Loránd University, Budapest, Hungary
{akaposi,vetuaat}@inf.elte.hu
[3] Heriot-Watt University, Scotland, UK
a.sinkarovs@hw.ac.uk

Abstract

In one of his tall tales, Baron Münchhausen pulled himself out of a swamp by his own hair. Inspired by this, we present a technique to justify "very dependent types": terms with types that include the term itself. The Münchhausen method is an informal way to make this precise. We don't have to resort to untyped preterms or typing relations, the technique works in a completely algebraic setting (such as categories with families). We present the method through a series of examples.

A dependent version of "products as functions from Bool"

There is a well known type isomorphism

$$A \times B \;\cong\; (b : \mathsf{Bool}) \to \mathsf{if}\, b\, \mathsf{then}\, A\, \mathsf{else}\, B.$$

Can we turn the nondependent product into a Σ type? Given $A : \mathsf{Type}$, $B : A \to \mathsf{Type}$, we want something like

$$\Sigma\, A\, B \;\cong\; (b : \mathsf{Bool}) \to \mathsf{if}\, b\, \mathsf{then}\, A\, \mathsf{else}\, (B\, \square),$$

but we don't know what to put in the placeholder $\square$. It should be the output of the function when the input is $b = \mathsf{true}$. Once the function is given a name, we can refer to it:

$$f : (b : \mathsf{Bool}) \to \mathsf{if}\, b\, \mathsf{then}\, A\, \mathsf{else}\, (B\, (f\, \mathsf{true}))$$

This is sometimes called a "very dependent type" [3]: the term f appears in its own type. It is possible to make sense of such a type using preterms and typing relations [1], but we can also present it algebraically as follows.

$$\begin{aligned} &a_0 : A \\ &f \;\; : (b : \mathsf{Bool}) \to \mathsf{if}\, b\, \mathsf{then}\, A\, \mathsf{else}\, (B\, a_0) \\ &\not{a}_0 : a_0 = f\, \mathsf{true} \end{aligned}$$

We first declare a_0 as the extra component that the type of f will depend on. Then we can declare f itself with the help of a_0. Finally, knowing about f, we can equate a_0 away: after learning about $\not{a}_0$, we know that the type of f is $(b : \mathsf{Bool}) \to \mathsf{if}\, b\, \mathsf{then}\, A\, \mathsf{else}\, (B\, (f\, \mathsf{true}))$. We only needed a_0 for bootstrapping the type of f.

*Ambrus Kaposi was supported by the "Application Domain Specific Highly Reliable IT Solutions" project which has been implemented with support from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme, and by Bolyai Scholarship BO/00659/19/3.

†This work is supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.

Type theory without types

A well-known example of the same technique is equating the sort of types away. We declare the sorts and operations of type theory as follows.

$$\begin{aligned}
&\mathsf{Con} : \mathsf{Type}\\
&\mathsf{Ty} \;\; : \mathsf{Con} \to \mathsf{Type}\\
&\mathsf{Tm} \;\; : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Type}\\
&\mathsf{U} \;\; : \mathsf{Ty}\,\Gamma\\
&\cancel{\mathsf{Ty}} \;\; : \mathsf{Ty}\,\Gamma = \mathsf{Tm}\,\Gamma\,\mathsf{U}
\end{aligned}$$

We have to introduce types, but once we have $\cancel{\mathsf{Ty}}$, we know that types are just terms of type U. Note that such a theory would be inconsistent through Russell's paradox, but it is easy to fix this by stratification (adding natural number indices to Ty and U, see [5]). Thus Münchhausen provides an algebraic way to define universes à la Russell, that is, a type theory without types: we use types only for bootstrapping.

Type theory without contexts

Just as we equated types away, we can do the same with contexts and substitutions. We start with the signature for categories with families (CwF [2]) with the four sorts $\mathsf{Con} : \mathsf{Type}$, $\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Type}$, $\mathsf{Ty} : \mathsf{Con} \to \mathsf{Type}$, $\mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Type}$, empty context $\diamond : \mathsf{Con}$, context extension $-\triangleright- : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con}$, and we have $\top$ and Σ types. Then we add equations such as $\mathsf{Con} = \mathsf{Ty}\,\diamond$, $\mathsf{Sub}\,\Delta\,\Gamma = \mathsf{Tm}\,\Delta\,(\Gamma[\epsilon])$, $\sigma\circ\delta = \sigma[\delta]$, $\Gamma\triangleright A = \Sigma\,\Gamma\,(A[\mathsf{q}])$. In the end we have e.g. $\mathsf{Tm} : (\Gamma : \mathsf{Ty}\,\top) \to \mathsf{Ty}\,\Gamma \to \mathsf{Type}$ and $\Sigma : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Sigma\,\Gamma\,(A[\mathsf{q}])) \to \mathsf{Ty}\,\Gamma$.

The language with all these equations can be justified: any CwF with $\top$ and Σ can be turned into an equivalent CwF which satisfies all these equations. There is an analogous model construction for type theory with types, and the two model constructions can be combined.

Towards type theory without contexts for real

Simply typed combinator calculus is a language where contexts are not even present for bootstrapping [4]. There are no variables, function space is built-in and the combinators S and K are used to define functions. Due to the technical challenges which come with not being able to use variables [6], combinator calculus was never extended to dependent types. We are in the process of defining a combinatory (dependent) type theory using Münchhausen's technique.

We first introduce types $\mathsf{Ty} : \mathsf{Type}$, terms indexed by types $\mathsf{Tm} : \mathsf{Ty} \to \mathsf{Type}$, a universe $\mathsf{U} : \mathsf{Ty}$ with the equation $\mathsf{Ty} = \mathsf{Tm}\,\mathsf{U}$, then we introduce families $- \Rightarrow\mathsf{U} : \mathsf{Ty} \to \mathsf{Ty}$ with instantiation $- \$ - : \mathsf{Tm}\,(A \Rightarrow\mathsf{U}) \to \mathsf{Tm}\,A \to \mathsf{Ty}$. Now we are in the position of declaring dependent function space $\Pi : (A : \mathsf{Ty}) \to \mathsf{Tm}\,((A \Rightarrow\mathsf{U}) \Rightarrow\mathsf{U})$ and application $- \cdot - : \mathsf{Tm}\,(\Pi\,A\,\$\,B) \to (a : \mathsf{Tm}\,A) \to \mathsf{Tm}\,(B\,\$\,a)$. We introduce constant families $\mathsf{K_f} : \mathsf{Ty} \to \mathsf{Tm}\,(B \Rightarrow\mathsf{U})$ with the equation $\mathsf{K_f}\,A\,\$\,b = A$ which allows us to express the non-dependent function type $A \Rightarrow B := \Pi\,A\,\$\,(\mathsf{K_f}\,B)$. Using a helper combinator $- \Rightarrow_\mathsf{K} - : \mathsf{Tm}\,(A \Rightarrow\mathsf{U}) \to \mathsf{Ty} \to \mathsf{Tm}\,(A \Rightarrow\mathsf{U})$ with equation $(B \Rightarrow_\mathsf{K} C)\,\$\,a = B\,\$\,a \Rightarrow C$, we can can express the dependent version of the K combinator $\mathsf{K} : \mathsf{Tm}\,(\Pi\,A\,\$\,B \Rightarrow_\mathsf{K} A)$ with equation $\mathsf{K}\cdot a\cdot b = a$. We can do the same for the dependently typed S combinator.

Our goal is to show that the syntax of combinatory type theory is equivalent to the syntax of CwF with $\top$, Σ, Π and universes.

References

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

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

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

[4] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

[5] András Kovács. Generalized universe hierarchies and first-class universe levels. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 28:1–28:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[6] Conor McBride. What is the combinatory logic equivalent of intuitionistic type theory? Answer to question on StackOverflow, 2012. https://stackoverflow.com/questions/11406786/what-is-the-combinatory-logic-equivalent-of-intuitionistic-type-theory.

The Patch Frame of a Spectral Locale in Univalent Type Theory

Ayberk Tosun and Martín H. Escardó

University of Birmingham, United Kingdom

Summary. Stone locales together with continuous maps form a coreflective subcategory of spectral locales and perfect maps. A proof in the internal language of an elementary topos is given in [1, 2]. This proof can be easily translated to univalent type theory using *resizing axioms*. In this work, we show how to achieve such a translation *without* resizing axioms, by working with large, locally small frames with small bases. This turns out to be nontrivial and involves predicative reformulations of several fundamental concepts of locale theory.

Foundations and notation. We work in the context of intensional Martin-Löf type theory without full univalence but with propositional and function extensionality and propositional truncation. We assume the existence of $\sum$ and $\prod$ types as well as the inductive types of $\mathbf{0}$, $\mathbf{1}$, the natural numbers, and lists. We define the type of families on a given type A as $\mathsf{Fam}_{\mathcal{W}}\,(A) :\equiv \sum_{I:\mathcal{W}} I \to A$. Given a family $(I, \alpha) : \mathsf{Fam}_{\mathcal{W}}\,(A)$, we often use the abbrevation $\{\alpha(i)\}_{i:I}$ instead of the tuple notation. Given a family $\mathcal{J} :\equiv (J, \beta)$ on the index type I, we write $\{\alpha(j) \mid j \in \mathcal{J}\}$ to denote the composite $(J, \alpha \circ \beta)$ i.e. the subfamily of $\{\alpha(i)\}_{i:I}$ given by (J, β).

Spectral and Stone locales. A *spectral locale* (also called *coherent* [4, pg. 63]) is a locale in which the compact opens form a basis closed under finite meets. A continuous map of spectral locales is perfect iff its defining frame homomorphism preserves compact opens. A *Stone locale* is one that is compact and zero-dimensional (i.e. whose clopens form a basis). Every Stone locale is spectral since the clopens coincide with the compact opens in Stone locales. We denote by **Stone** the category of Stone locales with continuous maps, and by **Spec** the category of Spectral locales with perfect maps. The right adjoint to the inclusion $\mathbf{Stone} \hookrightarrow \mathbf{Spec}$ is denoted by **Patch**.

Patch as the frame of Scott-continuous nuclei. A nucleus on a frame is a finite-meet-preserving closure operator. The patch frame of a spectral frame can be formulated as the frame of Scott-continuous nuclei on the frame [1]. A consequence of this description is that the patch frame freely adds Boolean complements to the given frame.

Locales with small bases. A $(\mathcal{U}, \mathcal{V}, \mathcal{W})$-frame is a type $A : \mathcal{U}$ equipped with (1) a partial order $- \leq - : A \to A \to \Omega_{\mathcal{V}}$, a top element $\mathbf{1} : A$, (2) a binary meet operation $- \wedge - : A \to A \to A$, and (3) a join operation $\bigvee - : \mathsf{Fam}_{\mathcal{W}}\,(A) \to A$ such that binary meets distribute over arbitrary joins:

$$\prod_{x:A} \prod_{(I,\alpha):\mathsf{Fam}_{\mathcal{W}}(A)} x \wedge \bigvee(I, \alpha) = \bigvee(I, \lambda i.\ x \wedge \alpha(i)).$$

We follow the standard convention of talking about locales, for which we use the variables $X, Y, Z, \ldots$, and referring to their frame of opens as $\mathcal{O}(X)$. A $(\mathcal{U}, \mathcal{V}, \mathcal{W})$-locale X is said to have a *small basis* iff there exists a $\mathcal{W}$-family $\{B_i\}_{i:I}$ on $\mathcal{O}(X)$ that satisfies:

$$\mathsf{isBasisFor}\,(\mathcal{B}, X) \quad :\equiv \quad \prod_{U:\mathcal{O}(X)} \sum_{J:\mathsf{Fam}_{\mathcal{W}}(I)} U \text{ is the least upper bound of } \{B_j \mid j \in J\}.$$

Spectrality, regularity, and zero-dimensionality. The notions of spectral, regular, and zero-dimensional locales are defined, in the impredicative setting of set theory, as locales in which certain kinds of opens form bases. A spectral locale, for example, is one in which the set of compact opens forms a basis closed under finite meets. Such definitions are problematic in a predicative context as it is not always the case that such sets of opens are small. We therefore restrict attention to locales with small bases and express these notions by imposing conditions on the bases in consideration. The notion of a spectral $(\mathcal{U}, \mathcal{V}, \mathcal{W})$-locale, for example, is defined as a locale with a small basis $\{B_i\}_{i:I}$ together with the requirements:

$$\prod_{i:I} B_i \text{ compact} \quad \text{and} \quad \prod_{i,j:I} \left\| \sum_{k:I} B_i \wedge B_j = B_k \right\|.$$

Same idea is employed in the definitions of regular and zero-dimensional locales. Another contribution concerns the predicative reformulations of locale-theoretic results about these notions, suitable for formalisation in type theory.

Construction of the patch frame. The strategy of [1] is to start from the known fact that the set of nuclei on a frame themselves form a frame, and then conclude that the Scott-continuous nuclei form a subframe. However, the first step does not seem to be available in our setting, and we need a different method of proof to show that the Scott-continuous nuclei form a frame. Similarly, other constructions in [1, 2] need to be completely rethought.

Open nuclei and AFT. In the impredicative setting, to any open U of a locale X, there is an associated open nucleus $\neg`U' :\equiv V \mapsto U \Rightarrow V$, where $- \Rightarrow -$ denotes Heyting implication, which is of fundamental importance in the construction of the patch. In the absence of resizing axioms, however, it does not seem to be possible in univalent type theory to construct Heyting implication for an arbitrary frame. Nevertheless, this is possible for locally small frames with small bases. More generally, we prove the Adjoint Functor Theorem for such frames in type theory: given a $(\mathcal{U}, \mathcal{V}, \mathcal{V})$-locale X with a small basis and a $(\mathcal{U}', \mathcal{V}, \mathcal{V})$-locale (not necessarily with a small basis), a monotone map $\mathcal{O}(Y) \to \mathcal{O}(X)$ has a right adjoint iff it preserves all joins of $\mathcal{O}(Y)$.

Formalisation. Most of our results have been formalised using the Agda proof assistant as part of the first author's `formal-topology-in-UF` library [5]. A more up-to-date formalisation is being developed as part of the second author's `TypeTopology` [3] library. Crucial components have already been formalised in modules `Frame`, `CompactRegular`, `GaloisConnection`, `AdjointFunctorTheoremForFrames`, and `HeytingImplication` of `TypeTopology`[1].

References

[1] Martín H. Escardó. "On the Compact-regular Coreflection of a Stably Compact Locale". In: *Electronic Notes in Theoretical Computer Science* 20 (1999), pp. 213–228. ISSN: 15710661. DOI: 10.1016/S1571-0661(04)80076-8.

[2] Martín H. Escardó. "The Regular Locally Compact Coreflection of a Stably Locally Compact Locale". In: *Journal of Pure and Applied Algebra* 157.1 (Mar. 8, 2001), pp. 41–55. ISSN: 0022-4049. DOI: 10.1016/S0022-4049(99)00172-3.

[1] Can be viewed on the `master` branch of `TypeTopology` at https://github.com/martinescardo/TypeTopology.

[3] Martín H. Escardó and contributors. *TypeTopology*. AGDA library. URL: https://github.com/martinescardo/TypeTopology.

[4] Peter T. Johnstone. *Stone Spaces*. Cambridge: Cambridge Univ. Press, 2002. ISBN: 978-0-521-33779-3.

[5] Ayberk Tosun. *formal-topology-in-UF*. AGDA library. Dec. 24, 2021. URL: https://github.com/ayberkt/formal-topology-in-UF.

Towards a denotational semantics of streams for a verified Lustre compiler

Timothy Bourke, Paul Jeanmaire, Marc Pouzet

DI ENS, École normale supérieure, Université PSL, CNRS, INRIA, Paris, France

Vélus [1] is a formally verified compiler for the Lustre synchronous programming language. It is developed in Coq and uses the CompCert C compiler as a back-end. The correctness theorem links the dataflow semantics of the source language to the semantics of the generated assembly code. Its proof is a composition of the individual proofs of each compilation pass.

For Vélus it has been proved that repeated execution of the generated assembly code faithfully implements the dataflow semantics of source programs. To facilitate the compilation correctness proof, the choice was made to model the input language with a relational-style semantics, as shown in the following statement.

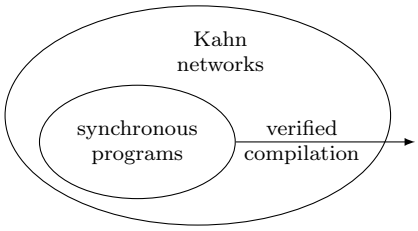
Theorem (compilation correctness, simplified)**.** *Given a Lustre node f, a list of input streams xs and a list of output streams ys, if f is successfully compiled to an assembly program P then:*

$$\mathsf{sem\_node}\ f\ xs\ ys \implies \exists T \in Traces(P),\ T \simeq (xs, ys).$$

Here the inductive predicate $\mathsf{sem\_node} : \mathsf{node} \to \mathsf{list\ Stream} \to \mathsf{list\ Stream} \to \mathsf{Prop}$ describes a relation whose elements are, for each node, the possible pairs of input and output streams. Since Lustre nodes denote deterministic stream functions, we were able to show that for all f, xs, ys and ys', $\mathsf{sem\_node}\ f\ xs\ ys \implies \mathsf{sem\_node}\ f\ xs\ ys' \implies ys \simeq ys'$.

While these definitions are well-tailored to establish compilation correctness, notably in the transition between dataflow and imperative languages, they do not give a procedure to build streams that satisfy the predicates. In particular, the determinism of nodes ensures that there is at most one possible output for a given input, but it does not guarantee the existence of such an output. Although unlikely, it could be the case that sem_node has no inhabitants, thus rendering void the main correctness theorem.

Since directly stating and proving the existence of a witness is very challenging due to the mutually recursive nature of equations in a Lustre node, it seems more appropriate to reason forward by defining a constructive interpretation of Lustre programs and then showing that the computed streams actually satisfy sem_node.



One possible approach is to consider the original definition of the language. The set of Lustre programs is naturally determined as a restricted class of Kahn networks [2] that can be executed *synchronously* and with bounded buffers. The ability to statically bound the required memory has long been exploited to design control software, especially in the certified development of safety-critical applications. In [3], Christine Paulin-Mohring describes how to

give a constructive denotational semantics to Kahn networks in Coq by means of a general library for CPOs, defining stream operations as the least fixed-points of continuous functions.

We are using this library to define a denotational semantics for Lustre. The aim is to provide a more natural front-end semantics for Vélus, closer to the one introduced in seminal articles [4]. There are two potential advantages to this approach. First, we believe it will facilitate the existence proof, because the the sem_node predicate is defined in a similar manner. A witness could also be constructed using a different style, for instance, using coiteration [5] which defines streams using iterated transition functions. We think, though, that it would be more difficult to relate the sequence global valuations so generated to the streams used in sem_node. There is also reason to believe that a denotational model à la Kahn may be the most convenient for interactively verifying Lustre programs that involve sampling since the absence of values is represented implicitly [6].

Mechanizing synchronous programs as Kahn networks in Coq challenges us to finely state the assumptions necessary to ensure that they compile correctly and execute safely. In the Kahn model, streams are built by iterating continuous stream functions, starting from the empty sequence. The first step is to ensure that the computed streams are indeed infinite, as required by sem_node. We are proving it by characterizing the class of *constructive* stream functions and exploiting a causality predicate required of source programs.

Finally, we obtain a denotation $[\![\cdot]\!]$ of Lustre components. For every node f, $[\![f]\!]$ is a total continuous function that maps infinite input streams to infinite output streams that may contain error values. We aim to show that these errors only arise from runtime exceptions (division by zero, integer overflow, etc.) which cannot be detected statically. We conjecture that if no such error occurs in output or local streams, then the relational predicate sem_node holds.

Conjecture (coherence of the relational semantics)**.** *Given a Lustre node f and a list of input streams xs, if $[\![f]\!](xs)$ is exempt of runtime errors then:*

$$\mathsf{sem\_node}\ f\ xs\ ([\![f]\!](xs)).$$

References

[1] T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for Lustre," in *PLDI 2017 - 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Barcelone, Spain), ACM, June 2017.

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974* (J. L. Rosenfeld, ed.), pp. 471–475, North-Holland, 1974.

[3] C. Paulin-Mohring, "A constructive denotational semantics for Kahn networks in Coq," in *From Semantics to Computer Science* (Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin, eds.), pp. 383–413, Cambridge University Press, 2009.

[4] P. Caspi and M. Pouzet, "Synchronous Kahn networks," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, vol. 31, 08 2001.

[5] P. Caspi and M. Pouzet, "A co-iterative characterization of synchronous stream functions," in *First Workshop on Coalgebraic Methods in Computer Science (CMCS'98)*, vol. 11 of *ENTCS*, (Lisbon, Portugal), pp. 1–21, Elsevier Science, Mar. 1998.

[6] C. Canovas-Dumas and P. Caspi, "A PVS proof obligation generator for Lustre programs," in *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings* (M. Parigot and A. Voronkov, eds.), vol. 1955 of *Lecture Notes in Computer Science*, pp. 179–188, Springer, 2000.

Towards a Formalization of Affine Schemes in Cubical Agda

Anders Mörtberg and Max Zeuner

Stockholm University, Stockholm, Sweden
{anders.mortberg,zeuner}@math.su.se

Schemes are the corner stone of modern algebraic geometry. Roughly speaking, they are topological spaces equipped with a sheaf of rings that locally look like so-called *affine schemes*. These affine schemes arise from central notions of commutative algebra: the *Zariski topology* defined on the set of prime ideals of a commutative ring R is equipped with a particular sheaf of rings, the *structure sheaf* using *localizations* of R. An early formalization of affine schemes in the `Coq` proof assistant can already be found in [4] but more recently, a full-blown formalization of general schemes was added to `Lean`'s `mathlib` [3]. By now, schemes have also been defined in `Isabelle/HOL` [2] and in `Coq`'s `UniMath`-library.[1] All of the aforementioned formalizations follow the inherently non-constructive approach of Hartshorne's classic textbook "Algebraic Geometry" [7] when defining the structure sheaf.

Working in `Cubical Agda`, an extension of the `Agda` proof assistant based on cubical type theory with fully constructive support of the univalence axiom [11], we want to give a *constructive* formalization of affine schemes in a univalent setting. This also seems to be in line with the aim of Voevodsky's Foundations library [12] to develop a library of constructive set-level mathematics based on a univalent foundation. To this end, we decided to follow the approach described by Coquand et. al. in [5]. This can be seen as a constructivization of the common approach that first defines the structure sheaf on the *basic opens*, a canonical basis of the Zariski topology, and then use some general machinery to extend this to a sheaf on the whole space.

Localization Localizations are generalized fractions and for the structure sheaf we need localizations at a single element: for a ring R and an element $f \in R$, the localization $R[1/f]$ corresponds roughly to the ring of fractions of the form x/f^n. Localizations can directly be defined as set-quotients, one of `Cubical Agda`'s higher inductive types (HIT). The universal property of localization gives us for two elements $f, g \in R$ a unique isomorphism of rings $R[1/f][1/g] \cong R[1/fg]$. When verifying the sheaf property of the structure sheaf, these two rings are however identified and freely substituted across the proof in informal mathematics. Using univalence, or more precisely `Cubical Agda`'s structure identity principle [1], we can promote this isomorphism to an equality $R[1/f][1/g] = R[1/fg]$. At this point our formalization differs already from the `Lean` formalization. Without univalence this equality is not obtainable, which led the authors of [3] to adapt a somewhat non-standard approach to localization.

Zariski Lattice The main difference between the constructive approach of [5] and classical ones is the use of a synthetic description of the Zariski lattice. We have to give a point-free definition of this lattice since prime ideals do not behave well constructively. Classically, open sets of the Zariski topology are generated by basic opens $D(f)$, the set of prime ideals of R that do *not* contain $f \in R$. The synthetic Zarsiki lattice is the free distributive lattice generated by *formal symbols* $D(f)$ quotiented by some relations that make these formal basic opens behave like their classical counterparts.

This synthetic definition was first given by Joyal in [8], but in our formalization we use a more explicit construction due to Español [6]: The Zariski lattice is formalized as `List` R, the

[1] See https://github.com/UniMath/UniMath/tree/master/UniMath/AlgebraicGeometry

type of lists with elements in R, quotiented by the relation

$$[\alpha_0, \ldots, \alpha_n] \sim [\beta_0, \ldots, \beta_m] \quad :\Leftrightarrow \quad \sqrt{\langle \alpha_0, \ldots, \alpha_n \rangle} = \sqrt{\langle \beta_0, \ldots, \beta_m \rangle}$$

This means that two lists are considered equal if the *radicals* of the ideals generated by their respective elements are equal. Since we are thus only working over finitely generated ideals, elements of this quotient are in bijective correspondence with open sets of the Zariski topology if R is *Noetherian*.

Equipping this set-quotient with the structure of a distributive lattice requires a lot of standard results about radical ideals and facts about $\_ + \_$ and $\_ \cdot \_$ of ideals, as those will give rise to the lattice operations. On top of that we need to introduce operations $\_ ++ \_$ and $\_ \cdot\cdot \_$ on lists that correspond to addition and multiplication of the ideals generated by those lists. For addition $\_ ++ \_$ is just list concatenation and for multiplication $\alpha \cdot\cdot \beta$ is the list of all products of the form $\alpha_i \beta_j$. The bottleneck then becomes proving that

$$\begin{aligned} \langle [\alpha_0, \ldots, \alpha_n] ++ [\beta_0, \ldots, \beta_m] \rangle &= \langle \alpha_0, \ldots, \alpha_n \rangle + \langle \beta_0, \ldots, \beta_m \rangle \\ \langle [\alpha_0, \ldots, \alpha_n] \cdot\cdot [\beta_0, \ldots, \beta_m] \rangle &= \langle \alpha_0, \ldots, \alpha_n \rangle \cdot \langle \beta_0, \ldots, \beta_m \rangle \end{aligned}$$

Structure Sheaf In the above implementation of the Zariski lattice, the basic open $D(f)$ corresponds to the equivalence class of the singleton list $[f]$. In HoTT/UF [10], subsets of a type X are functions from X into hProp, the universe of (homotopy) propositions. So if we denote by L_R the Zariski lattice associated with R, the basic opens are a function $B_R : L_R \to \mathsf{hProp}$ mapping $\alpha \in L_R$ to

$$B_R(\alpha) \quad := \quad \exists_{f:R} \left(D(f) = \alpha \right) \quad := \quad \| \Sigma_{f:R} \left(D(f) = \alpha \right) \|$$

Here $\|\_\|$ is the propositional truncation, another HIT in `Cubical Agda`, turning any type into a proposition. The general strategy is now to construct the structure sheaf on basic opens and then use some general results from category theory to obtain a sheaf on the whole Zariski lattice. The construction of the presheaf on basic opens as well as a weak form of the sheaf property, are fully formalized.[2] The full sheaf property and its lift to the whole lattice are currently work in progress.

For defining the basic presheaf we need to give a map $\left(\Sigma_{\alpha:L_R} B_R(\alpha)\right) \to \mathsf{CommRing}$ from elements of the Zariski lattice that are basic opens into the type of commutative rings, mapping $D(f)$ to $R[1/f]$. Here we run into a problem resulting from working in a univalent setting: Given an α s.t. $B_R(\alpha)$, the information which $D(f)$ equals to α is hidden under a propositional truncation while the goal type $\mathsf{CommRing}$ is a *groupoid* (i.e. a 1-type). To eliminate this truncation we need to appeal to a result by Kraus [9]. This however requires us to check some higher coherence condition, i.e. construct square fillers in $\mathsf{CommRing}$. The key observation at this point is that the structure sheaf on L_R does actually take values in R-Alg, the type of R-algebras, as localizations of R are always R-algebras. The ring-valued structure sheaf can then be obtained by composing with the forgetful functor R-$\mathsf{Alg} \to \mathsf{CommRing}$. Some standard commutative algebra tells us that for $f, g \in R$ with $\sqrt{\langle f \rangle} \subseteq \sqrt{\langle g \rangle}$ there is a *unique* homomorphism of R-algebras $R[1/g] \to R[1/f]$.

It turns out that this not only solves the coherence issues, but actually facilitates the presheaf construction and the proof of the (weak) sheaf property by a rather neat argument that makes essential use of the primitives of `Cubical Agda` as well as the equality $R[1/f][1/g] = R[1/fg]$, while avoiding cumbersome diagram chases.

[2] See https://github.com/agda/cubical/tree/master/Cubical/Algebra/ZariskiLattice

References

[1] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. Internalizing representation independence with univalence. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[2] Anthony Bordg, Lawrence Paulson, and Wenda Li. Simple type theory is not too simple: Grothendieck's schemes without dependent types. arXiv preprint, 2021. https://arxiv.org/abs/2104.09366.

[3] Kevin Buzzard, Chris Hughes, Kenny Lau, Amelia Livingston, Ramon Fernández Mir, and Scott Morrison. Schemes in lean. *Experimental Mathematics*, 0(0):1–9, 2021.

[4] Laurent Chicli. Une formalisation des faisceaux et des schémas affines en théorie des types avec Coq. Technical Report RR-4216, INRIA, June 2001.

[5] Thierry Coquand, Henri Lombardi, and Peter Schuster. Spectral schemes as ringed lattices. *Annals of Mathematics and Artificial Intelligence*, 56(3):339–360, 2009.

[6] Luis Español. Le spectre d'un anneau dans l'algèbre constructive et applications à la dimension. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 24(2):133–144, 1983.

[7] Robin Hartshorne. *Algebraic geometry*, volume 52. Springer Science & Business Media, 2013.

[8] André Joyal. Les théoremes de chevalley-tarski et remarques sur l'algèbre constructive. *Cahiers Topologie Géom. Différentielle*, 16:256–258, 1976.

[9] Nicolai Kraus. The general universal property of the propositional truncation. In Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau, editors, *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 111–145, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[11] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31:e8, 2021.

[12] Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015.

Towards a Mechanized Theory of Computation for Education

Tiago Cogumbreiro[1] and Yannick Forster[2]

[1] University of Massachussets Boston, Boston, Massachussets, U.S.A.
tiago.cogumbreiro@umb.edu
[2] Inria, France
yannick.forster@inria.fr

Abstract

In this talk we present a mechanization effort of theory of computation in the context of undergraduate education, with a focus on decidability and computability. We introduce a Coq library used to teach 3 sessions of a course on Formal Languages and Automata, at the University of Massachusetts Boston. Our project includes full proofs of results from a textbook, such as the undecidability of the halting problem and Rice's theorem. To this end, we present a simple and expressive calculus that allows us to capture the essence of informal proofs of classic theorems in a mechanized setting. We discuss the assumptions of our formalism and discuss our progress in showing the consistency of our theory.

Introduction. Formal languages and automata (FLA) is in the basis of the curriculum of undergraduate computer science [10]. We report on an open source project written in Coq [3] to mechanize results of classical theory of computation. The first author used this software for 3 semesters to teach decidability, computability, and regular languages at the University of Massachusetts Boston. Proof assistants play a central role in our lectures for three reasons. Firstly, a proof assistant offers an interactive mechanism to allow students to step through a proof autonomously, allowing students to independently browse every detail of a proof at their own pace. Secondly, a proof assistant turns a logic assignment into a programming assignment, which can be more approachable to computer science students. Thirdly, having proof scripts that can be machine checked, lets instructors automatically grade homework assignments. Other works that use proof assistants to aid education include [1, 9, 12].

Mechanization goals. We formalize Sipser's *Introduction to the theory of computation* [16] in Coq. Our design goal is to keep our formalism as close to the textbook as possible, which includes having mechanized proofs that mirror the textbook proofs. Another important design goal is that proofs should only include basic Coq capabilities. The proofs need to be comprehensible to an undergraduate student with rudimentary knowledge of Coq (case analysis, induction, polymorphism, and logical connectives). Further, we include alternative proofs of some theorems when there's a pedagogical benefit, *e.g.*, the proof is simpler, or the intuition is easier to explain. Our approach contrasts many published works on mechanized computability theory [13, 17, 8, 2, 15, 4, 11].

Decidability results. Our mechanization includes the main results of [16, Chapters 4 and 5], on decidability and reducibility. One of our contributions is formalizing Sipser's "high-level descriptions," which is essentially pseudo-code to describe a Turing machine. For instance, consider Theorem 4.11 of [16, Chapter 4], where TM denotes a Turing machine, ranged over by meta-variable M, inputs are ranged over by i. A language, ranged over by A is a set of inputs. Language A is decidable if there exists a Turing machine M that decides A, *i.e.*, M accepts i if, and only if, $i \in A$; and M rejects i if, and only if, $i \notin A$. Additionally, $\langle \cdot \rangle$ denotes a reasonable encoding of one or more objects into a string, *e.g.*, $\langle M, i \rangle$ encodes a Turing machine M and an input i into an input, and $\langle M \rangle$ encodes a Turing machine M into an input.

Theorem 4.11 ([16, pp. 207]). *$A_{TM} = \{\langle M, i\rangle \mid M$ is a TM and M accepts $i\}$ is undecidable.*

The proof of Theorem 4.11 includes the following high-level description of a Turing machine D, parameterized by a Turing machine H which decides A_{TM}: *"The following is the description of D: (1) Run H on input $\langle M, \langle M\rangle\rangle$. (2) Output the opposite of what H outputs. That is, if H accepts,* reject*; and, if H rejects, accept."*

We formalize such high-level description as:

```
Definition D (H:input → prog): input → prog :=
  fun (i:input) ⇒                          (* On input i = <M> *)
    mlet b ← H <[ decode_mach i, i ]> in (* Step 1 *)
    if b then Ret false else Ret true    (* Step 2 *)
```

where input $\texttt{i} = \langle M\rangle$ and $\texttt{decode\_mach i} = M$. We formalize high-level descriptions next. We first give the syntax of high-level descriptions p.

$$p \ ::= \ \texttt{mlet}\ x = p\ \texttt{in}\ p \ \mid\ \texttt{call}\ M\ i \ \mid\ \texttt{return}\ b \qquad \text{where } b \in \{\top, \bot\}$$

Next, we introduce a big-step operational semantics in terms of high-level descriptions:

$$\frac{}{\texttt{return}\ b \Downarrow b} \qquad \frac{M \text{ accepts } i}{\texttt{call}\ M\ i \Downarrow \top} \qquad \frac{M \text{ rejects } i}{\texttt{call}\ M\ i \Downarrow \bot} \qquad \frac{p \Downarrow b \qquad p'[x := b] \Downarrow b'}{\texttt{mlet}\ x = p\ \texttt{in}\ p' \Downarrow b'}$$

We then define that a Turing machine M computes a Turing function f of type $\texttt{input} \to \texttt{prog}$ if for any input i we have $\texttt{call}\ M\ i \Downarrow b$ if and only if $f(i) \Downarrow b$.

Assumptions. Our theory is parameterized by an input type, a type of Turing machines, and the semantics of Turing machines. We assume that their execution is deterministic and that for any machine M and an input i we can obtain M accepts i, or M rejects i, or neither. Centrally, we assume that for any Turing function f there exists a Turing machine M computing f. This assumption is consistent since we work in Coq, where every definable function is computable.

Results. To mechanize the proof of Theorem 4.11 we assume a machine M' deciding A_{TM}, *i.e.*, computing a Turing function H. Now using the assumption that every Turing function is computable on $\texttt{D}(H)$ yields M such that $\texttt{call}\ M\ i \Downarrow \texttt{true} \leftrightarrow \texttt{call}\ M\ i \Downarrow \texttt{false}$, a contradiction.

Our further main results include: A is decidable if, and only if A is recognizable and co-recognizable (*i.e.*, its complement is recognizable) (Theorem 4.22); the complement of A_{TM} is not recognizable (Corollary 4.23); $HALT_{TM} = \{\langle M, i\rangle \mid M$ accepts or rejects $i\}$ is undecidable (Theorem 5.1); $E_{TM} = \{\langle M\rangle \mid L(M) = \emptyset\}$ is undecidable (Theorem 5.2), where $L(M) = \{i \mid M \text{ accepts } i\}$; $EQ_{TM} = \{\langle M_1, M_2\rangle \mid L(M_1) = L(M_2)\}$ is neither recognizable nor co-recognizable (Theorem 5.30); EQ_{TM} is undecidable (Theorem 5.4); Rice's Theorem (Problem 5.28). Our proofs of these results are all constructive. Our project contains results on languages and regular languages, *e.g.*, the pumping lemma for regular language inspired by the proof in [14].

Future work. To show the consistency of our axioms, we are working on instantiating our theory with a mechanized formalism of computability from the Coq library of undecidability proofs [7] equivalent to Turing machines [6]. Consistency of the central assumptions then follows from the consistency of the axiom CT in type theory [5]. We are also investigating multiple grading approaches in classes that use proof assistants, *e.g.*, multiple-choice questions, automatic questions about students submissions.

References

[1] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol PaźK, and Josef Urban. Mizar: State-of-the-art and beyond. In *CICM*, volume 9150, page 261–279. Springer, 2015.

[2] Mario Carneiro. Formalizing Computability Theory via Partial Recursive Functions. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] Tiago Cogumbreiro. Turing: formalization of introduction to the theory of computation, 2022. https://gitlab.com/umb-svl/turing.

[4] Yannick Forster. Parametric church's thesis: Synthetic computability without choice. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science - International Symposium, LFCS 2022, Deerfield Beach, FL, USA, January 10-13, 2022, Proceedings*, volume 13137 of *Lecture Notes in Computer Science*, pages 70–89. Springer, 2022.

[5] Yannick Forster. Parametric Church's Thesis: Synthetic computability without choice. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science - International Symposium, LFCS 2022, Deerfield Beach, FL, USA, January 10-13, 2022, Proceedings*, volume 13137 of *Lecture Notes in Computer Science*, pages 70–89. Springer, 2022.

[6] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified programming of Turing machines in Coq. In *CPP*, page 114–128. ACM, 2020.

[7] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*., 2020.

[8] Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017.

[9] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. Teaching a formalized logical calculus. In *ThEdu*, volume 313 of *EPTCS*, pages 73–92, 2019.

[10] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, 2013.

[11] J Strother Moore. Towards a mechanically checked theory of computation. In *Logic-Based Artificial Intelligence*, pages 547–574. Springer US, 2000.

[12] Tobias Nipkow. Teaching semantics with a proof assistant: No more LSD trip proofs. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *LNCS*, pages 24–38. Springer, 2012.

[13] Michael Norrish. Mechanised computability theory. In *ITP 2011*, volume 6898 of *LNCS*, pages 297–311. Springer, 2011.

[14] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2021. Version 6.1. https://softwarefoundations.cis.upenn.edu/lf-current/.

[15] Thiago Mendonça Ferreira Ramos, Ariane Alves Almeida, and Mauricio Ayala-Rincón. Formalization of the computational theory of a turing complete functional language model. *Journal of Automated Reasoning*, January 2022.

[16] Michael Sipser. *Introduction to the theory of computation*. Cengage Learning, 2012.

[17] Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing machines and computability theory in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 147–162. Springer, 2013.

Towards a translation from $\mathbb{K}$ to Dedukti

Amélie Ledein[1*], Valentin Blot[1], Catherine Dubois[2]

[1] Laboratoire Méthodes Formelles, Inria, Université Paris-Saclay
[2] Samovar, ENSIIE

In this talk, we present a shallow embedding of the $\mathbb{K}$ formalism allowing the definition of programming languages semantics, in the $\lambda\Pi$-calculus modulo theory. We propose a solution that makes use of the intermediate representation provided by the $\mathbb{K}$ compiler, i.e. Kore and relies on the rewriting engine of Dedukti in order to be able to execute a program in the translated formalism as it is possible to do with the execution tool provided by the $\mathbb{K}$ framework.

$\mathbb{K}$ presentation. $\mathbb{K}$ [2] is a semantical framework for formally describing the semantics of programming languages. It is also an environment that offers various tools to help programming with the languages specified in the formalism (Figure 1). It is for example possible to execute programs and to check some properties on them, using the automatic theorem KProver tool [10]. $\mathbb{K}$ is based on a theory of Matching Logic [9, 10, 6, 5, 4], named Kore, a 1st order untyped classic logic with an application between formulas and, fixed point, equality and typing operators, as well as an operator similar to the "next" operator of temporal logics. This last operator allows the semantics of programs to be encoded by rewriting. Kore is composed of the equality theory, the sort theory and the rewriting one.

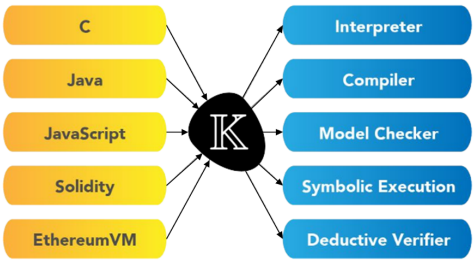


Figure 1: Pipeline of $\mathbb{K}$

Dedukti presentation. Dedukti [3] is a logical framework à la LF allowing the interoperability of proofs between different formal proof tools, as Coq or PVS (Figure 2). It has import and export plugins for proof systems as various as Coq, PVS or Isabelle/HOL. Dedukti is based on the $\lambda\Pi$-calculus modulo theory ($\lambda\Pi\equiv_{\mathcal{T}}$), an extension of the type theory by adding rewriting rules [8] in the conversion relation, introduced by Cousineau and Dowek [7]. The flexibility of this logical framework allows to encode many theories like 1st order logic or simple type theory.

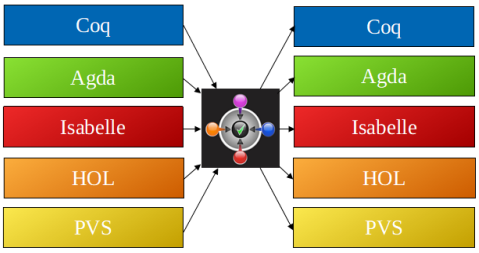


Figure 2: Pipeline of Dedukti

*The author is funded by Digicosme.

Contributions. In this talk, we present KaMeLo [1], a tool for translating from 𝕂 to Dedukti. This tool has the longer term goal of allowing the verification of proofs made within 𝕂, and the reuse of formal semantics of programming languages as well as properties of their programs in many proof systems, via Dedukti.

𝕂 is a rewrite-based framework in which programming languages semantics can be defined using configurations, computations and rules. Formally, a configuration is a multi-set of potentially nested labelled cells, e.g. a cell containing a program to be execute, another corresponding to an environment and a last one which is a memory store. Rewriting rules specify the evolution of a configuration during the execution. 𝕂 provides the user with many facilities to write a semantic definition, e.g. attributes to define a left to right evaluation strategy or an ellipsis like ... to indicate that there is no change in the rest of a configuration. However whatever the style used in the 𝕂 definition, in the Kore produced file everything is a plain and more standard form. Our translation from 𝕂 to Dedukti, is done via Kore, to make use of this *standardisation* that we assume correct.

𝕂 and Dedukti have the common point of being both based on rewriting, where this can be applied anywhere in a term, and can be non-linear. However, Dedukti allows higher order, while 𝕂 supports only 1st order. Conversely, conditional rewriting and modulo ACUI are not supported by Dedukti, unlike 𝕂. These rewriting rules are preserved in Dedukti. The next two paragraphs deal with more specific cases concerning the translation of rewriting rules.

Concerning conditional rewriting the general idea of the encoding, inspired from [11], is, for a symbol defined with conditional rules, to generate a fresh symbol with the same arguments as the initial symbol, plus as many arguments as there are conditions. Once these conditional arguments have been instantiated, it is possible to evaluate them, and then to rewrite the term as a whole according to the evaluation of these arguments.

Whatever the way used to define an evaluation strategy in 𝕂, conditional rewriting rules will be generated, using a *K computation*, i.e. a potentially nested list of computations to be performed sequentially, and *freezers*. Intuitively, a freezer is a symbol which encapsulate the part of the computation that shouldn't modify, i.e. the queue of the K computation, while waiting for the head of the K computation to be evaluated. These generated rules have the particularity of not being able to be transformed thanks to our variant of Viry's algorithm, under penalty of losing the confluence property. In this particular case, we have opted for the specialisation of the left-hand side terms of the rules, i.e. to refine the pattern-matching in order to precise the desired type of the left-hand side terms of the rules. To do this, we use the subtyping relations, but also the inductive structure.

Conclusion & Perspectives. Thanks to the logical framework Dedukti, the objective of this work is to verify formal proofs about the semantics of programming languages, described in the semantical framework 𝕂, and to reuse of such proofs in different proof tools.

Initially, we were interested in the translation in Dedukti of semantics written in 𝕂, in order to be able to execute in Dedukti programs written in the language described by the semantics. We have relied on the theory of Matching Logic, named Kore, without seeking to certify it. We therefore assume that the Kore file produced from a 𝕂 semantic is correct. This work required understanding the translation of a 𝕂 semantics into Kore but also being able to translate conditional rules into unconditional rules. For now, the translator KaMeLo, a tool for translating 𝕂 to Dedukti, currently under development, translates the syntax of the language, the configurations and symbols, some attributes and the rewriting rules.

The verification of the KProver proof objects, as well as the encoding of the theoretical foundations of 𝕂 in those of Dedukti will be future work. The translation presented here is nevertheless necessary to run a program and will be reused for the verification of proofs.

References

[1] GitLab of KaMeLo. https://gitlab.com/semantiko/kamelo.

[2] Website of $\mathbb{K}$. https://kframework.org/.

[3] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Expressing theories in the $\lambda\Pi$-calculus modulo theory and in the Dedukti system. In *TYPES: Types for Proofs and Programs*, Novi SAd, Serbia, May 2016.

[4] X. Chen, D. Lucanu, and G. Roşu. Matching Logic Explained. Technical Report http://hdl.handle.net/2142/107794, University of Illinois at Urbana-Champaign and Alexandru Ioan Cuza University, July 2020.

[5] X. Chen and G. Roşu. Applicative Matching Logic: Semantics of K. Technical Report http://hdl.handle.net/2142/104616, University of Illinois at Urbana-Champaign, July 2019.

[6] X. Chen and G. Rosu. Matching mu-Logic: Foundation of K Framework (Invited Paper). page 4 pages, 2019. Artwork Size: 4 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany Version Number: 1.0.

[7] D. Cousineau and G. Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *TLCA*, 2007.

[8] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320, 1990.

[9] G. Roșu and T. F. Șerbănuță. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, Aug. 2010.

[10] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 74–91, Amsterdam Netherlands, Oct. 2016. ACM.

[11] P. Viry. Elimination of Conditions. *Journal of Symbolic Computation*, 28(3):381–401, 1999.

Towards Higher Observational Type Theory

Thorsten Altenkirch[1], Ambrus Kaposi[2]*, and Michael Shulmann[3]†

[1] School of Computer Science, University of Nottingham, UK
psztxa@nottingham.ac.uk
[2] Eötvös Loránd University, Budapest, Hungary
{akaposi}@inf.elte.hu
[3] University of San Diego, USA
shulman@sandiego.edu

Abstract

Do we need to go cubical when presenting Homotopy Type Theory? The seminal paper by Cohen, Coquand, Huber and Mörtberg [4] presented important progress in giving not only a constructive semantics of Homotopy Type Theory (which was first achieved by Bezem, Coquand, and Huber [3]) but also in using this model to present a computationally well behaved Type Theory which lead to implementations like Cubical Agda [7] and RedTT. (Although the notions of "cubes" used in all these cases are slightly different). The basic idea here is to introduce a special type, called the interval, and model equality as paths, i.e. as functions from the interval to the given type. A composition operator which corresponds to Kan filling operations can then be defined by recursion over types and most importantly we can give a constructive interpretation of univalence exploiting the fact that the interval is tiny, i.e. exponentiation with it has a right adjoint. The latter property isn't easily transferable to other models, e.g. simplicial sets, leading to a mismatch with standard constructions in homotopy theory.

In the present work we attempt to find a different way to formulate a Type Theory with univalence, which we call Higher Observational Type Theory. The basic idea is to avoid the introduction of an interval but instead to provide rules how to calculate equality types for every type former. To achieve this we first introduce a calculus of logical relations, which forces us to consider context extensions called telescopes. We present this telescopic calculus using higher order abstract syntax, which means that all constructions should be viewed as taking place in the presheaf category over contexts and substitutions, i.e. all rules are relative to a context, all type formers are functorial, and all term formers are natural transformations.

1 Equality

We write telescopes as Δ **Tel**; these are extensions of a given context Γ. A telescopic substitution $\delta : \Delta$ is basically a substitution Γ to $\Gamma\Delta$ which is inverse to the projection. We introduce the usual judgements relative to a telescope $\Delta \vdash \mathcal{J}$ and they are closed under telescopic substitution: that is from the above we can derive $\mathcal{J}[\delta]$.

*Ambrus Kaposi was supported by the "Application Domain Specific Highly Reliable IT Solutions" project which has been implemented with support from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme, and by Bolyai Scholarship BO/00659/19/3.

†Michael Shulman was supported by the United States Air Force Office of Scientific Research under award number FA9550-21-1-0009.

We introduce a homogeneous equality on telescopes and a heterogenous equality on types:

$$\frac{\Delta\ \mathbf{Tel} \qquad \delta_0, \delta_1 : \Delta}{\mathrm{Id}_\Delta\, \delta_0\, \delta_1\ \mathbf{Tel}} \qquad \frac{\Delta \vdash A : \mathsf{U} \qquad \delta_2 : \mathrm{Id}_\Delta\, \delta_0\, \delta_1 \qquad a_0 : A[\delta_0] \qquad a_1 : A[\delta_1]}{\mathrm{Id}^{\delta_2}_{\Delta.A}\ a_0\ a_1 : \mathsf{U}}$$

Note that all these rules are in a presheaf setting, i.e. relative to a given context Γ. The telescope equality is composed of the type equality: $\mathrm{Id}_{\Delta\, A}\, (\delta_0\, a_0)\, (\delta_1\, a_1) \equiv (\delta_2 : \mathrm{Id}_\Delta\, \delta_0\, \delta_1)\, (\mathrm{Id}^{\delta_2}_{\Delta.A}\ a_0\ a_1)$. Every telescopic substitution and every term preserve equality:

$$\frac{\Delta \vdash A : \mathsf{U} \qquad \delta_0, \delta_1 : \Delta \qquad \delta_2 : \mathrm{Id}_\Delta\, \delta_0\, \delta_1 \qquad \Delta \vdash a : A}{\mathrm{ap}^{\delta_2}_{\Delta.a} : \mathrm{Id}^{\delta_2}_{\Delta.A}\, a[\delta_0]\, a[\delta_1]}$$

Now homogeneous equality and reflexivity arise as special cases:

$$\frac{A : \mathsf{U} \qquad a_0, a_1 : A}{\mathrm{Id}_A\, a_0\, a_1 : \mathsf{U}} \qquad \mathrm{Id}_A\, a_0\, a_1 \equiv \mathrm{Id}^{()}_{\bullet.A}\, a_0\, a_1$$

$$\frac{A : \mathsf{U} \qquad a : A}{\mathrm{refl}_A\, a : \mathrm{Id}_A\, a\, a : \mathsf{U}} \qquad \mathrm{refl}_A\, a \equiv \mathrm{ap}^{()}_{\bullet.a}$$

2 Univalence

We define equality in the universe to be a relational characterisation of equivalence such as that in [6, Exercise 4.2]. A proof relevant relation (or "correspondence") $R : A_0 \to A_1 \to \mathsf{U}$ is an equivalence (or "one-to-one correspondence"), written $\mathrm{Eq}\, R$, if it is a function in both directions. It is a function from left to right if $\Sigma a_1 : A_1.R\, a_0\, a_1$ is contractible for all $a_0 : A_0$.

$$\frac{\delta_0, \delta_1 : \Delta \qquad \delta_2 : \mathrm{Id}_\Delta\, \delta_0\, \delta_1 \qquad A_0, A_1 : \mathsf{U}}{\mathrm{Id}^{\delta_2}_{\Delta.\mathsf{U}}\, A_0\, A_1 \equiv \Sigma(R : A_0 \to A_1 \to \mathsf{U})\, \mathrm{Eq}\, R}$$

Now equality is just given by the relation we obtain from type equality. That means we can actually compute equality using ap:

$$\frac{\Delta \vdash A : \mathsf{U} \qquad \delta_0, \delta_1 : \Delta \qquad \delta_2 : \mathrm{Id}_\Delta\, \delta_0\, \delta_1 \qquad a_0 : A[\delta_0] \qquad a_1 : A[\delta_1]}{\mathrm{Id}^{\delta_2}_{\Delta.A}\ a_0\ a_1 \equiv \pi_1\, \mathrm{ap}^{\delta_2}_{\Delta.A}\, a_0\, a_1}$$

That means that the only thing we need to define is ap. Given this definition of equality in the universe we can define weak path induction (that is without the definitional β-equality).

In particular we need to show that the universe is closed under the standard type formers, i.e. that they preserve equivalences. The lifting of the relation to type formers is the standard lifting of logical relations.

3 Progress so far

This is work in progress and hence incomplete and subject to change. It is to some degree inspired by earlier work by the first two authors [1]. A more concise definition of the calculus can be given in terms of Categories with Families (CwF) in a presheaf category. We note that the telescope calculus gives rise to a CwF in the presheaf category and that the congruence rules we give arise from the notion of logical relations as given by [2]. While our definition of the universe seems sound, it is not clear whether this approach gives rise to a straightforward decision procedure. Another line of research is to show that some cubical set model can be presented in a way to satisfy the definitional equalities of our calculus; and more generally that it can be interpreted in model categories for all higher toposes, as was shown for non-computational Homotopy Type Theory in [5].

References

[1] Thorsten Altenkirch and Ambrus Kaposi. Towards a cubical type theory without an interval. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 3:1–3:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[2] Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 135–144. IEEE Computer Society, 2012.

[3] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs*, volume 26 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages 107–128. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2014.

[4] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.

[5] Michael Shulman. All $(\infty, 1)$-toposes have strict univalent universes. arXiv:1904.07004, 2019.

[6] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book/, first edition, 2013.

[7] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.*, 31:e8, 2021.

Towards Probabilistic Reasoning about Typed Combinatory Terms

Simona Kašterović[1] and Silvia Ghilezan[1,2]

[1] Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
simona.k@uns.ac.rs, gsilvia@uns.ac.rs
[2] Mathematical Institute of the Serbian Academy of Sciences and Arts, Belgrade, Serbia

The interest in probabilistic programming has increased over the past decades due to the role which reasoning with uncertainty has in computer science and artificial intelligence. The developments in computer science and artificial intelligence urge for formalizing uncertain reasoning. Combinatory logic found its application in computer science as a model of computation. In order to formalize reasoning with uncertainty about programs, we propose probabilistic model for reasoning about typed combinatory terms.

We present results of [8] and ongoing work that emerged from [4, 3, 8]. In [8] we introduced *Logic of Combinatory Logic* (*LCL*), a classical propositional logic for reasoning about simply typed combinatory logic. *LCL* is obtained by defining the classical propositional logic over simply typed combinatory logic. The language of *LCL* is defined by the following grammar:

$$\alpha := M : \sigma \mid \neg\alpha \mid \alpha \wedge \alpha,$$

where $M : \sigma$ is type assignment statement typable from some basis Γ in simply typed combinatory logic, M is a combinatory term and σ is a simple type.

The axiomatic system of *LCL* is obtained from the axiomatic system for classical propositional logic and the type assignment system for simply typed combinatory logic (Figure 1). Instances of axiom schemes can only be built up from formulas of the language, hence, $M : \sigma$ can be a subformula of some instance of an axiom scheme only if $M : \sigma$ is type assignment statement typable from some basis Γ in simply typed combinatory logic. Besides the axiom schemes of classical propositional logic and the inference rule Modus Ponens, the proposed axiomatic system contains three non-logical axiom schemes for typing primitive combinators $\mathsf{S}, \mathsf{K}, \mathsf{I}$ and two axiom schemes that correspond to the typing rules of simply typed combinatory logic. The simple type system developed in [2] does not have an equality rule which would correspond to (Ax 5). However, in [5] Hindley has added this rule in order to obtain completeness of the type assignment system for lambda calculus. In the notation $=_{w,\eta}$, w denotes that it is a transitive, reflexive and symmetric closure of the one-step reduction, and η denotes that it is an extensional relation.

We proposed semantics for *LCL*, inspired by Kripke-style semantics for lambda calculus with types introduced in [9, 7]. The semantics for *LCL* is based on an extensional applicative structure containing special elements that correspond to primitive combinators. The proposed semantics are not a Kripke-style semantics as the ones presented in [9, 7], still the definition of the applicative structure is inspired by the definition of the applicative structure presented in [9, 7]. An applicative structure for *LCL* is a tuple $\mathcal{M} = \langle D, \{A^\sigma\}_\sigma, \cdot, \mathbf{s}, \mathbf{k}, \mathbf{i}\rangle$ where D is a non-empty set, called domain; $\{A^\sigma\}_\sigma$ is a family of subsets of the domain D; $\cdot$ is an extensional binary operation on D such that $\cdot : A^{\sigma\to\tau} \times A^\sigma \to A^\tau$ for every $\sigma, \tau \in \mathsf{Types}$; $\mathbf{s}, \mathbf{k}, \mathbf{i}$ are special elements of the domain D that correspond to the primitive combinators, that is $\mathbf{s} \in A^{(\sigma\to\tau\to\rho)\to(\sigma\to\tau)\to(\sigma\to\rho)}$ and $((\mathbf{s}\cdot a)\cdot b)\cdot c = (a\cdot c)\cdot(b\cdot c)$, and similar for the elements $\mathbf{k}, \mathbf{i}$.

We have proved that the given axiomatic system is sound and complete with respect to the proposed semantics.

Axiom schemes:

(Ax 1) $\mathsf{S} : (\sigma \to (\tau \to \rho)) \to ((\sigma \to \tau) \to (\sigma \to \rho))$
(Ax 2) $\mathsf{K} : \sigma \to (\tau \to \sigma)$
(Ax 3) $\mathsf{I} : \sigma \to \sigma$
(Ax 4) $(M : \sigma \to \tau) \Rightarrow ((N : \sigma) \Rightarrow (MN : \tau))$
(Ax 5) $M : \sigma \Rightarrow N : \sigma$, if $M =_{w,\eta} N$
(Ax 6) $\alpha \Rightarrow (\beta \Rightarrow \alpha)$
(Ax 7) $(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$
(Ax 8) $(\neg\alpha \Rightarrow \neg\beta) \Rightarrow ((\neg\alpha \Rightarrow \beta) \Rightarrow \alpha)$

Inference rule: $\dfrac{\alpha \Rightarrow \beta \qquad \alpha}{\beta}$ (MP)

Figure 1: Axiom schemes and inference rule for LCL

Our goal is to develop a formal model for probabilistic reasoning about simply typed combinatory terms. In [4, 3], formal models are introduced for probabilistic reasoning about simply typed lambda terms and lambda terms with intersection types, respectively. These models are based on the well-known semantics for typed lambda calculus, namely, term models for simply typed lambda calculus ([5]) and filter models for lambda calculus with intersection types ([1]). However, these models are not well-suited for propositional reasoning about typed terms, thus we develop a model for probabilistic reasoning about typed combinatory terms based on the Logic of Combinatory Logic ([8]).

We propose a probabilistic system for simply typed combinatory terms, PCL which is a probabilistic logic ([6, 10]) over LCL. We extend LCL with the probabilistic operator $P_{\geq s}$, and obtain a system expressive enough to write formulas of the form $P_{\geq s}\alpha$ which has a meaning "probability that α is true is greater than or equal to s". The language of PCL consists of two sets of formulas: *basic formulas* and *probabilistic formulas*. Basic formulas are formulas of LCL. Probabilistic formulas are formulas generated by the following grammar

$$\phi := P_{\geq s}\alpha \mid \neg\phi \mid \phi \wedge \phi,$$

where α is an LCL-formula and $s \in [0, 1] \cap \mathbb{Q}$.

We propose Kripke-style semantics, where LCL-models serve as possible worlds. To interpret probabilities we equip the set of possible worlds with a probability measure. We will give an infinitary axiomatic system, obtained from the axiomatic system of LCL along with the axiomatic system for probability logic.

In order to obtain soundness and completeness results for PCL we had to prove soundness and completeness results on the level of basic formulas. Kripke-style semantics for the proposed logic is built from models for LCL, by defining probability measure over the set of LCL-models. Thus, the completeness result for LCL ([8]) will play a key role in proving completeness of the given axiomatization with respect to the proposed semantics.

Acknowledgements We wish to thank the anonymous reviewers for their valuable suggestions.

References

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

[2] Haskell Brooks Curry and Robert M. Feys. *Combinatory Logic Vol. 1.* Amsterdam, Netherlands: North-Holland Publishing Company, 1958.

[3] Silvia Ghilezan, Jelena Ivetić, Simona Kašterovć, Zoran Ognjanović, and Nenad Savić. Towards probabilistic reasoning in type theory - the intersection type case. In Andreas Herzig and Juha Kontinen, editors, *Foundations of Information and Knowledge Systems - 11th International Symposium, FoIKS 2020, Dortmund, Germany, February 17-21, 2020, Proceedings*, volume 12012 of *Lecture Notes in Computer Science*, pages 122–139. Springer, 2020.

[4] Silvia Ghilezan, Jelena Ivetić, Simona Kašterović, Zoran Ognjanović, and Nenad Savić. Probabilistic reasoning about simply typed lambda terms. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science - International Symposium, LFCS 2018, Deerfield Beach, FL, USA, January 8-11, 2018, Proceedings*, volume 10703 of *Lecture Notes in Computer Science*, pages 170–189. Springer, 2018.

[5] J. Roger Hindley. The completeness theorem for typing lambda-terms. *Theor. Comput. Sci.*, 22:1–17, 1983.

[6] Nebojša Ikodinović, Zoran Ognjanović, Aleksandar Perović, and Miodrag Rašković. Hierarchies of probabilistic logics. *Int. J. Approx. Reason.*, 55(9):1830–1842, 2014.

[7] Simona Kašterović and Silvia Ghilezan. Kripke-style semantics and completeness for full simply typed lambda calculus. *J. Log. Comput.*, 30(8):1567–1608, 2020.

[8] Simona Kašterović and Silvia Ghilezan. Logic of combinatory logic, February 2021. Submitted for publication.

[9] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Ann. Pure Appl. Log.*, 51(1-2):99–124, 1991.

[10] Zoran Ognjanović, Miodrag Rašković, and Zoran Marković. *Probability Logics - Probability-Based Formalization of Uncertain Reasoning.* Springer, 2016.

Transpension: The Right Adjoint to the Pi-Type

Andreas Nuyts[1] and Dominique Devriese[1]

imec-DistriNet, KU Leuven, Belgium

Presheaf models of dependent type theory [Hof97, HS97] have been successfully applied to model HoTT [BCH14, CMS20, CCHM17, Hub16, KLV12, Ort18, OP18], parametricity [AGJ14, BCM15, ND18a, NVD17], and directed, guarded [BM20] and nominal [Pit13, §6.3] type theory, as well as combinations of these [BBC+19, CH20, RS17, WL20].[1] If we want to reap the fruit of such models *within* type theory and be able to write proofs that are unsound in simpler models, then we need internal operators reflecting some aspects of the model. While the constructions of presheaf models for various applications largely follow a common pattern, approaches towards internalization do not. Throughout the literature, various internal presheaf operators (the amazing right adjoint $\surd$ [LOPS18], Φ/extent and Ψ/Gel [BCM15, Mou16, CH20], Glue and Weld [CCHM17, NVD17], mill [ND18b], the strictness axiom [OP18] and locally fresh names [PMD15]) can be found with little or no analysis of their relative expressiveness. Understanding the common foundations of these operators would provide guidance to developers of new type systems, and may allow for cross fertilization between existing systems. For example, understanding that the Gel-type for parametricity – which so far has only been formulated w.r.t. an affine interval – is closely related to $\surd$ – which has been formulated w.r.t. the cartesian interval of cubical HoTT – can help us to generalize both to other flavours of intervals.

Three years ago [ND19], we proposed the *transpension* type $\Diamond u : \mathrm{Ty}(\Gamma) \to \mathrm{Ty}(\Gamma, u : \mathbb{U})$, right adjoint to structural or substructural universal quantification $\forall(u : \mathbb{U}) : \mathrm{Ty}(\Gamma, u : \mathbb{U}) \to \mathrm{Ty}(\Gamma)$ over a shape $\mathbb{U}$ (such as the interval), as a means of internalizing the peculiarities of presheaf models in general. Each of the aforementioned internal operators can be implemented from transpension, strictness and/or a pushout type former.[2] The transpension type has a structure reminiscent of a dependent version of the suspension type in HoTT [Uni13, §6.5]. In topoi, a right adjoint to structural quantification $\Pi(u : \mathbb{U})$ has already been considered by Yetter [Yet87], who named it ∇ and proved that it is definable from the amazing right adjoint $\surd$.

The structural transpension coquantifier $\Diamond u$ is part of a sequence of adjoints $\Sigma\, u \dashv \Omega\, u \dashv \Pi\, u \dashv \Diamond u$, preceded by the Σ-type, weakening and the Π-type. Adjointness of the first three is provable from the structural rules of type theory, but it is not immediately clear how to add typing rules for a further adjoint. Birkedal et al. [BCM+20] explain how to add a single modality that has a left adjoint in the semantics. If we want to have two or more adjoint modalities internally, then we can use a multimodal type system such as MTT [GKNB21, GKNB20].

In a paper currently under review [ND21] we present an extensional type system extending MTT, which features the transpension type as a modality and is backed by a presheaf model. Each internal modality in MTT needs a semantic left adjoint, so we can only internalize $\Omega\, u \dashv \Pi\, u \dashv \Diamond u$. A drawback which we accept (as a challenge for future work), is that $\Omega\, u$ and $\Pi\, u$ become modalities which are a bit more awkward to deal with than ordinary weakening and Π-types. Below, we explain the main ideas of our approach, without reiterating the benefits and applications of the transpension type [ND19].

Shapes and Multipliers. Transpension is right adjoint to universal quantification over a shape. Because we want to support both structural (cartesian) and substructural (e.g. affine) quantification, our definition of shape needs to be a bit more general than just 'a representable object' of the presheaf model $\mathrm{Psh}(\mathcal{W})$ which would be essentially, via the Yoneda-embedding, an object of $\mathcal{W}$. Instead, a shape $\mathbb{U}$ will be modelled by an arbitrary endofunctor denoted $\sqcup \ltimes U : \mathcal{W} \to \mathcal{W}$,[3] dubbed a *multiplier*. We write U for a chosen object isomorphic to $\top \ltimes U$ so that we can always project $\pi_2 : W \ltimes U \to U$.

[1] We omit models that are not explicitly structured as presheaf models [AHH18, LH11, Nor19].

[2] For locally fresh names, we only have a heuristic translation.

[3] In a technical report [Nuy21], we generalize beyond *endo*functors.

Examples of such shapes are: the interval $\mathbb{I}$ in affine and cartesian cubical models of HoTT and/or parametricity, the sort of names in nominal type theory [PMD15], the sort of clocks of a given finite longevity in guarded type theory [BM20], and the twisted prism functor [PK20] which we believe is important for directed type theory.

Because arbitrary endofunctors are a bit too general to obtain many useful results, we introduce some criteria to classify multipliers. The multiplier gives rise to a functor $\lrcorner_U : \mathcal{W} \to \mathcal{W}/U : W \mapsto (W \ltimes U, \pi_2)$ to the slice category over U. We say that $\sqcup \ltimes U$ is

- *semicartesian* if it is copointed, i.e. there is a first projection $\pi_1 : W \ltimes U \to W$,
- *cartesian* if it is a cartesian product,
- *cancellative* if $\lrcorner_U$ is faithful (equivalently if $\sqcup \ltimes U$ is),
- *affine* if $\lrcorner_U$ is full (which rules out being cartesian unless $U \cong \top$),
- *connection-free* if $\lrcorner_U$ is essentially surjective on slices (V, φ) such that $\varphi : V \to U$ is *dimensionally split*, which in most cases just means split epi,
- *quantifiable* if $\lrcorner_U$ has a left adjoint $\exists_U : \mathcal{W}/U \to \mathcal{W}$ (i.e. if $\sqcup \ltimes U$ is a local right adjoint).

For the properties of the example multipliers, we refer to the paper [ND21].

Modes are Shape Contexts. Every MTT judgement $p \mid \Gamma \vdash J$ is stated at some mode p, and modalities $\mu : p \to q$ have a domain and a codomain mode. The introduction rule for the modal type looks like this:

$$\frac{p \mid \Gamma, \blacksquare_\mu \vdash a : A \qquad \mu : p \to q}{q \mid \Gamma \vdash \mathsf{mod}_\mu\, a : \langle \mu \mid A \rangle}$$

Typically (but not necessarily) every mode p will be modelled by a presheaf category $[\![p]\!]$ and every modality $\mu : p \to q$ will be modelled by a DRA [BCM+20] $[\![\blacksquare_\mu]\!] \dashv [\![\mu]\!] : [\![p]\!] \to [\![q]\!]$.

A complication is that the modalities that we need, bind or depend on a variable, a phenomenon which is not supported by MTT. We solve this by grouping shape variables such as $u : \mathbb{U}$ in a *shape context* which is not considered part of the type-theoretic context but instead serves as the *mode* of the judgement. Formally, we define a shape context as any presheaf over $\mathcal{W}$, but in practice shape contexts will be denoted $(u_1 : \mathbb{U}_1, \ldots, u_n : \mathbb{U}_n)$ and obtained by applying (the left Kan extensions of) the corresponding multipliers to the terminal presheaf. Judgements in shape context Ξ are then interpreted in presheaves over the category of elements $\mathcal{W}/\Xi$, i.e. dependent presheaves over Ξ.

Modalities. As modalities $\mu : \Xi_1 \to \Xi_2$, we take *all* DRAs from $\mathrm{Psh}(\mathcal{W}/\Xi_1)$ to $\mathrm{Psh}(\mathcal{W}/\Xi_2)$. Again, a few specific ones are of special interest:

Modalities for substitution. A shape substitution (presheaf morphism) $\sigma : \Xi_1 \to \Xi_2$ leads to a functor $\Sigma^{/\sigma} : \mathcal{W}/\Xi_1 \to \mathcal{W}/\Xi_2$ which, by left Kan extension, precomposition and right Kan extension, leads to a triple of adjoint functors $\Sigma^{\sigma|} \dashv \Omega^{\sigma|} \dashv \Pi^{\sigma|} : \mathrm{Psh}(\mathcal{W}/\Xi_1) \to \mathrm{Psh}(\mathcal{W}/\Xi_2)$, the latter two of which can be internalized as modalities $\Omega\,\sigma \dashv \Pi\,\sigma$. Of course $\Omega\,\sigma$ is the substitution modality. In case σ is really a weakening $\pi : (\Xi, u : \mathbb{U}) \to \Xi$ over a semicartesian shape $\mathbb{U}$, then we write $\Omega\,u \dashv \Pi(u : \mathbb{U})$ and these stand for weakening and the Π-type.

Modalities for (co)quantification. A quantifiable multiplier gives rise to functors $\exists_U^{/\Xi} \dashv \lrcorner_U^{/\Xi} : \mathcal{W}/\Xi \to \mathcal{W}/(\Xi, u : \mathbb{U})$, whence by Kan extension and precomposition a quadruple of adjoint functors $\exists_U^{\Xi|} \dashv \lrcorner_U^{\Xi|} \dashv \forall_U^{\Xi|} \dashv \between_U^{\Xi|} : \mathrm{Psh}(\mathcal{W}/\Xi) \to \mathrm{Psh}(\mathcal{W}/(\Xi, u : \mathbb{U}))$, the latter three of which can be internalized as $\lrcorner\,u \dashv \forall(u : \mathbb{U}) \dashv \between\,u$ which stand for fresh weakening, substructural quantification and transpension.

Theorem. If a multiplier $\sqcup \ltimes U$ is:

1. semicartesian, then we get morphisms $\mathsf{spoil}_u : \lrcorner\,u \Rightarrow \Omega\,u$ and (hence) $\mathsf{cospoil}_u : \Pi\,u \Rightarrow \forall\,u$,
2. cartesian, then spoil_u and (hence) $\mathsf{cospoil}_u$ are isomorphisms,
3. cancellative and affine, then $\forall\,u \circ \lrcorner\,u \cong 1$ and (hence) $\forall\,u \circ \between\,u \cong 1$,
4. cancellative, affine and connection-free, then the transpension type admits a pattern-matching eliminator not unlike that of the suspension type; equivalently, Φ/extent is then sound.

Acknowledgements We want to thank Jean-Philippe Bernardy, Dan Licata and Andrea Vezzosi for inspiring discussions on the subject. Andreas Nuyts holds a Postdoctoral Fellowship from the Research Foundation - Flanders (FWO). This work was partially supported by a research project of the Research Foundation - Flanders (FWO).

References

[AGJ14] Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *Principles of Programming Languages*, 2014. doi:10.1145/2535838.2535852.

[AHH18] 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, *Computer Science Logic (CSL 2018)*, volume 119 of *LIPIcs*, pages 6:1–6:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9673, doi:10.4230/LIPIcs.CSL.2018.6.

[BBC+19] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. Guarded cubical type theory. *Journal of Automated Reasoning*, 63(2):211–253, 8 2019. doi:10.1007/s10817-018-9471-7.

[BCH14] Marc Bezem, Thierry Coquand, and Simon Huber. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, Dagstuhl, Germany, 2014. URL: http://drops.dagstuhl.de/opus/volltexte/2014/4628, doi:10.4230/LIPIcs.TYPES.2013.107.

[BCM15] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electron. Notes in Theor. Comput. Sci.*, 319:67 – 82, 2015. doi:http://dx.doi.org/10.1016/j.entcs.2015.12.006.

[BCM+20] Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science*, 30(2):118–138, 2020. doi:10.1017/S0960129519000197.

[BM20] Ales Bizjak and Rasmus Ejlers Møgelberg. Denotational semantics for guarded dependent type theory. *Math. Struct. Comput. Sci.*, 30(4):342–378, 2020. doi:10.1017/S0960129520000080.

[CCHM17] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017. URL: http://www.cse.chalmers.se/~simonhu/papers/cubicaltt.pdf.

[CH20] Evan Cavallo and Robert Harper. Internal parametricity for cubical type theory. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*, pages 13:1–13:17, 2020. doi:10.4230/LIPIcs.CSL.2020.13.

[CMS20] Evan Cavallo, Anders Mörtberg, and Andrew W Swan. Unifying Cubical Models of Univalent Type Theory. In Maribel Fernández and Anca Muscholl, editors, *Computer Science Logic (CSL 2020)*, volume 152 of *LIPIcs*, pages 14:1–14:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/11657, doi:10.4230/LIPIcs.CSL.2020.14.

[GKNB20] Daniel Gratzer, Alex Kavvos, Andreas Nuyts, and Lars Birkedal. Type theory à la mode. Pre-print, 2020. URL: https://anuyts.github.io/files/mtt-techreport.pdf.

[GKNB21] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal Dependent Type Theory. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021. URL: https://lmcs.episciences.org/7713, doi:10.46298/lmcs-17(3:11)2021.

[Hof97] Martin Hofmann. *Syntax and Semantics of Dependent Types*, chapter 4, pages 79–130. Cambridge University Press, 1997.

[HS97] Martin Hofmann and Thomas Streicher. Lifting grothendieck universes. Unpublished note, 1997. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf.

[Hub16] Simon Huber. *Cubical Interpretations of Type Theory*. PhD thesis, University of Gothenburg, Sweden, 2016. URL: http://www.cse.chalmers.se/~simonhu/misc/thesis.pdf.

[KLV12] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. 2012. Preprint, http://arxiv.org/abs/1211.2851.

[LH11] Daniel R. Licata and Robert Harper. 2-dimensional directed type theory. *Electr. Notes Theor. Comput. Sci.*, 276:263–289, 2011. doi:10.1016/j.entcs.2011.09.026.

[LOPS18] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal universes in models of homotopy type theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 22:1–22:17, 2018. doi:10.4230/LIPIcs.FSCD.2018.22.

[Mou16] Guilhem Moulin. *Internalizing Parametricity*. PhD thesis, Chalmers University of Technology, Sweden, 2016. URL: publications.lib.chalmers.se/records/fulltext/235758/235758.pdf.

[ND18a] Andreas Nuyts and Dominique Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In *Logic in Computer Science (LICS) 2018, Oxford, UK, July 09-12, 2018*, pages 779–788, 2018. doi:10.1145/3209108.3209119.

[ND18b] Andreas Nuyts and Dominique Devriese. Internalizing Presheaf Semantics: Charting the Design Space. In *Workshop on Homotopy Type Theory / Univalent Foundations*, 2018. URL: https://hott-uf.github.io/2018/abstracts/HoTTUF18_paper_1.pdf.

[ND19] Andreas Nuyts and Dominique Devriese. Dependable atomicity in type theory. In *TYPES*, 2019. URL: https://lirias.kuleuven.be/retrieve/540872.

[ND21] Andreas Nuyts and Dominique Devriese. Transpension: The right adjoint to the pi-type. *CoRR*, abs/2008.08533, 2021. URL: https://arxiv.org/abs/2008.08533, arXiv:2008.08533.

[Nor19] Paige Randall North. Towards a directed homotopy type theory. *Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2019, London, UK, June 4-7, 2019*, pages 223–239, 2019. doi:10.1016/j.entcs.2019.09.012.

[Nuy21] Andreas Nuyts. The transpension type: Technical report. *CoRR*, abs/2008.08530, 2021. URL: https://arxiv.org/abs/2008.08530, arXiv:2008.08530.

[NVD17] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *PACMPL*, 1(ICFP):32:1–32:29, 2017. URL: http://doi.acm.org/10.1145/3110276, doi:10.1145/3110276.

[OP18] Ian Orton and Andrew M. Pitts. Axioms for modelling cubical type theory in a topos. *Logical Methods in Computer Science*, 14(4), 2018. doi:10.23638/LMCS-14(4:23)2018.

[Ort18] Ian Orton. *Cubical Models of Homotopy Type Theory - An Internal Approach*. PhD thesis, University of Cambridge, 2018.

[Pit13] Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*, volume 57. Cambridge University Press, 2013.

[PK20] 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. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13069, doi:10.4230/LIPIcs.TYPES.2019.5.

[PMD15] Andrew M. Pitts, Justus Matthiesen, and Jasper Derikx. A dependent type theory with abstractable names. *Electronic Notes in Theoretical Computer Science*, 312:19 – 50, 2015. Ninth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2014). URL: http://www.sciencedirect.com/science/article/pii/S1571066115000079, doi:https://doi.org/10.1016/j.entcs.2015.04.003.

[RS17] E. Riehl and M. Shulman. A type theory for synthetic ∞-categories. *ArXiv e-prints*, May 2017. arXiv:1705.07442.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, IAS, 2013.

[WL20] Matthew Z. Weaver and Daniel R. Licata. A constructive model of directed univalence in bicubical sets. 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 915–928. ACM, 2020. doi:10.1145/3373718.3394794.

[Yet87] David Yetter. On right adjoints to exponential functors. *Journal of Pure and Applied Algebra*, 45(3):287–304, 1987. URL: https://www.sciencedirect.com/science/article/pii/0022404987900776, doi:https://doi.org/10.1016/0022-4049(87)90077-6.

Two Guarded Recursive Powerdomains for Applicative Simulation

Rasmus Ejlers Møgelberg and Andrea Vezzosi

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

Last year at Types we presented Clocked Cubical Type Theory (CCTT) [5], a type theory combining multi-clocked guarded recursion with Cubical Type Theory. One use case for this type theory is for programming and reasoning with coinductive types, encoding these via guarded recursion. CCTT allows one to do this also for coinductive types defined using higher inductive types, and one can moreover prove that path type equality for these coincides with bisimilarity. Another use is as a meta-language for both operational and denotational models of programming languages. This talk presents a worked example of using both these ideas, and is based on our newly published paper [9].

Guarded Powerdomains in Clocked Cubical Type Theory

Clocked Cubical Type Theory extends Cubical Type Theory with a pre-type of clocks, and for each clock κ, a modality $\triangleright^\kappa$ and a fixed point combinator $\mathsf{fix}^\kappa : (\triangleright^\kappa A \to A) \to A$. One use of the fixed point combinator is to construct guarded recursive types, such as $\mathsf{L}^\kappa A$ satisfying $\mathsf{L}^\kappa A \simeq A + \triangleright^\kappa \mathsf{L}^\kappa A$ as fixed points of maps on the universe. Defining a κ-delay algebra to be a type B with an operations $\triangleright^\kappa B \to B$, $\mathsf{L}^\kappa A$ is the free κ-delay algebra on A. Using quantification over clocks, one can use these to encode coinductive types such as $\mathsf{L}A \stackrel{\mathsf{def}}{=} \forall\kappa.\mathsf{L}^\kappa A$ which is the coinductive solution to $\mathsf{L}A \simeq A + \mathsf{L}A$. This latter is the coinductive delay monad, which can be used to model recursion in type theory, however, working with the guarded recursive variant L^κ gives access to a powerful fixed point operator, and, moreover, guarded recursive types can also have negative occurrences. This has been used for a form of guarded synthetic domain theory, producing models of FPC and PCF and proving these adequate in type theory [8, 11].

This work studies the extension of such models with finite non-determinism. We construct two guarded recursive powerdomains by combining L^κ with the finite powerset functor $\mathsf{P_f}$, defined as a HIT [3], generated by singleton, union and axioms making it the free join-semilatice. The powerdomains are defined as follows

$$\mathsf{P}^\kappa_\Diamond(A) \simeq \mathsf{P_f}(A + \triangleright^\kappa \mathsf{P}^\kappa_\Diamond(A)) \qquad\qquad \mathsf{P}^\kappa_\Box(A) \stackrel{\mathsf{def}}{=} \mathsf{L}^\kappa(\mathsf{P_f}(A))$$

The first of these is a monad defined as a guarded recursive type. An element of this type is a finite set of values of type A and computations that can be run for at least one more step. The subscript refers to may-convergence and is intuitively justified by the fact that it reveals return values for terminated branches even when other branches have not yet terminated. The second is simply the composition of two monads. Unlike $\mathsf{P}^\kappa_\Diamond$, elements of $\mathsf{P}^\kappa_\Box$ do not reveal partial results, but just returns a set of values once all branches have terminated. Unfortunately, $\mathsf{P}^\kappa_\Box$ is not a monad, since the associativity axiom breaks up to step counting. It is, however, a monad up to a notion of weak bisimilarity.

Semantics for the untyped lambda calculus

Both these constructions carry a semilattice structure. In the case of $\mathsf{P}^\kappa_\Diamond$ the union operation is defined using the one for P_f. In the case of $\mathsf{P}^\kappa_\Box$, the union operations evaluates the two given computations in parallel and returns the union once they have both terminated. This means that the delay is the maximum of the delays of the two input computations. Algebraically, $\mathsf{P}^\kappa_\Diamond(A)$ is the free join semilattice and κ-delay algebra on A with no equations between the two structures. For $\mathsf{P}^\kappa_\Box(A)$, the delay algebra structure distributes over the semilattice one, but also satisfies additional non-algebraic interaction equations.

Using the semilattice structures, one can define denotational semantics for the untyped lambda calculus extended with finite non-determinism in the form of an operation $M \,\mathsf{or}\, N$. In both cases, the domain of the denotational semantics is a solution to a guarded recursive domain equation defined as

$$\mathsf{SVal}^\kappa \stackrel{\mathsf{def}}{=} \triangleright^\kappa(\mathsf{SVal}^\kappa \to T(\mathsf{SVal}^\kappa)) \qquad\qquad D^\kappa \stackrel{\mathsf{def}}{=} T(\mathsf{SVal}^\kappa)$$

where T can be instantiated to $\mathsf{P}^\kappa_\Diamond$ and $\mathsf{P}^\kappa_\Box$ (or indeed any monad-like construction with a semilattice structure). This semantics can be proved sound with respect to the standard big-step may- and must operational semantics which we write as $\Downarrow_\Diamond$ and $\Downarrow_\Box$ respectively.

Applicative similarity

As an example application of these powerdomains, we look at how to prove applicative similarity a congruence for the untyped lambda calculus with finite non-determinism. This is usually proved using operational reasoning and Howe's method [7, 6, 4, 2], or in some cases advanced domain theoretic techniques such as Stone duality [10, 1]. Here we build on a proof by Pitts [12], which uses a denotational semantics in domain theory and a relation between syntax and semantics. Our contribution is to extend from the case of pure lambda calculus to finite non-determinism and adapt to guarded synthetic domain theory.

In a few more details, a relation R is an applicative may-simulation if $M\,R\,N$ and $M \Downarrow_\Diamond \lambda x.M'$ implies

$$\exists N'.\, N \Downarrow_\Diamond \lambda y.N' \wedge (\forall(V : \mathsf{Val}).M'[V/x]\,R\,N'[V/x])$$

May-similarity $\leq_\Diamond$ is the greatest may-simulation, and this can be defined in Clocked Cubical Type Theory using a combination of guarded recursion and quantification over clocks, similarly to the coinductive delay monad L. The aim is to show that this is a congruence. Our proof uses a relation $\preceq^\kappa$: $D^\kappa \times \Lambda \to \mathsf{Prop}$ between the denotational semantics mentioned in the previous section, and syntax. The key lemmas state that $[\![M]\!]^\kappa \preceq^\kappa M$ for all closed M, and that $M \leq_\Diamond N$ is equivalent to $\forall\kappa.[\![M]\!]^\kappa \preceq^\kappa N$. We prove a similar result for the case of must-similarity.

Implementation

The results mentioned above have been proved on paper, and only few lemmas have been formally verified in a proof assistant. In the time since this work was completed, Vezzosi has implemented an experimental extension of Agda[1] based on CCTT. Using this it should now be possible to implement these proofs in Agda without much overhead from the paper versions developed in this work. One point of the talk is therefore to announce the Agda branch for CCTT to the Types community.

[1] https://github.com/agda/guarded/tree/forcing-ticks

References

[1] Samson Abramsky. The lazy lambda calculus, research topics in functional programming. 1990.

[2] Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. Effectful applicative bisimilarity: Monads, relators, and howe's method. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.

[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] Douglas J Howe. Equality in lazy computation systems. In *LICS*, volume 89, pages 198–203, 1989.

[5] Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. Greatest hits: Higher inductive types in coinductive definitions via induction under clocks, 2021. arXiv:2102.01969.

[6] Søren B Lassen and Corin S Pitcher. Similarity and bisimilarity for countable non-determinism and higher-order functions. *Electronic Notes in Theoretical Computer Science*, 10:246–266, 1998.

[7] Søren Bøgh Lassen. *Relational reasoning about functions and nondeterminism.* PhD thesis, University of Aarhus, 1998.

[8] Rasmus Ejlers Møgelberg and Marco Paviotti. Denotational semantics of recursive types in synthetic guarded domain theory. In *LICS*, 2016.

[9] Rasmus Ejlers Møgelberg and Andrea Vezzosi. Two guarded recursive powerdomains for applicative simulation. In Ana Sokolova, editor, Proceedings 37th Conference on *Mathematical Foundations of Programming Semantics,* Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021, volume 351 of *Electronic Proceedings in Theoretical Computer Science*, pages 200–217. Open Publishing Association, 2021.

[10] C-HL Ong. Non-determinism in a functional setting. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 275–286. IEEE, 1993.

[11] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. A model of pcf in guarded type theory. *Electronic Notes in Theoretical Computer Science*, 319:333–349, 2015.

[12] Andrew M Pitts. A note on logical relations between semantics and syntax. *Logic Journal of the IGPL*, 5(4):589–601, 1997.

Type Inference via Symbolic Environment Transformations*

Ulrich Schöpp and Chuangjie Xu

fortiss GmbH, Munich, Germany

Region type systems are a powerful tool for e.g. pointer analysis and taint analysis [BGH13, GHL12] for object oriented programs. Extended with effect annotations, they can capture information about the possible event traces of programs and thus can be used to enforce programming guidelines [EHZ17, ESX21]. The idea of such a type-based analysis approach is to infer the type of a program which allows us to verify if the program satisfies certain properties. The type inference algorithms of [BGH13, GHL12, EHZ17, ESX21] infer a type for a method from the given types of its arguments. If the method is called with different arguments in different occasions of a program, then it is analyzed multiple times, one for each invocation.

To avoid analyzing the same piece of code multiple times, we introduce a *compositional* inference algorithm. Once a method has been analyzed, the result can be used directly in the analysis of its caller, assuming that it does not depend on the caller, i.e., it does not (indirectly) call the caller. The idea is to split the type inference into two steps: (1) Compute an environment transformation for each method. (2) Derive the type of the targeting method using its environment transformation. When analyzing a new method, we use the previously computed environment transformations of the callees rather than fully reanalyzing them as in previous work [BGH13, GHL12, EHZ17, ESX21]. We explain the idea in more detail.

Types and typing environments For simplicity, we work with a region type system without effects for Featherweight Java [IPW01] like the one in [BGH13]. A *type* is a region representing a property of a value such as its provenance information. For example, we may consider a region $\texttt{CreatedAt}(\ell)$ for references to objects that were created in the position with label ℓ. One can think of the label ℓ as a line number in the source code. This region allows us to track where in the program an object originates. Differing from previous work, we do not maintain a global table of field typing but instead add it into *typing environments*. For example, the environment

$$E = (x : \texttt{CreatedAt}(\ell_1),\ \texttt{CreatedAt}(\ell_1).f : \texttt{CreatedAt}(\ell_2))$$

means that x points to an object which is created at position ℓ_1 and the field f of any object created at ℓ_1 is an object created at ℓ_2.

Environment transformations The type system essentially performs flow analysis. The execution of a program may change the types of its variables and fields. Therefore, we can assign it an *environment transformation* that approximates how the types are updated in the program. For example, we assign the program

$$\begin{aligned} &y = x.f;\\ &x = \texttt{new}^{\ell_3}\ C(); \end{aligned}$$

the transformation

$$[y \mapsto x.f,\ x \mapsto \texttt{CreatedAt}(\ell_3)].$$

It updates the above environment E to $(x : \texttt{CreatedAt}(\ell_3),\ y : \texttt{CreatedAt}(\ell_2),\ \texttt{CreatedAt}(\ell_1).f : \texttt{CreatedAt}(\ell_2))$. Note that the substitutions are performed simultaneously.

*Supported by the German Research Foundation (DFG) under research grant 250888164 (GuideForce).

Field access graphs Directly using field access paths like $x.f$ as above is problematic, because the lengths of access paths may be unbounded. The computation of environment transformations involving such access paths may not terminate. For example, consider a class of linked lists with a field `next` : `Node` pointing the next node. The following method returns the last node of a list.

```
Node last() {
  if (next == null) {return this;}
  else {return next.last();}
}
```

It would have return type $\texttt{this} \vee \texttt{this.next} \vee \texttt{this.next.next} \vee \texttt{this.next.next.next} \vee \cdots$, expressing that the returned value has the same type of the variable `this` or the field `this.next` and so on. To solve this, we work with *access graphs* which provide a finite representation of access paths [KSK07]. For example, the `Node` class has three access graphs to represent all its access paths. Therefore, the return type of `last`() is the disjunction of these three graphs rather than the above infinite disjunction of access paths.

A theory of abstract transformations With the above ingredients, we now define a notion of abstract transformation. Like the usual ones, an *abstract transformation* consists of finitely many assignments $\kappa \mapsto u$. The value u is a disjunction of some atoms. An *atom* is a variable, a type or a field graph following a variable or a type. The key κ is a non-type atom, as we treat variables and fields similarly in order to involve field typing in environments. To model how types are updated in a program, we define the following operations on abstract transformations. We define the *composition* of abstract transformations to model type updates in a statement followed by another and the *join* of abstract transformations to tackle conditional branches. Moreover, we *instantiate* an abstract transformation to an endofunction on typing environments. Assignments are essentially subtyping constraints [PS91] and the instantiation solves the constraints.

Type inference Suppose we have a table T assigning an abstract transformation to each method of a program. Then we can compute an abstract transformation for any expression e of the program by induction on e. For example, when e is an invocation of a method, we lookup the table T. For any well-typed program, we can compute such a table T for it as follows. We start by initializing T for each method with the "bottom" transformation, that is, the transformation whose values are the empty disjunction. For each method, we compute the abstract transformation of its body, and then update the corresponding entry in T. We repeat this until no more update of T is possible. This procedure always terminates, because the update of T "weakens" its entries and there are only finitely many abstract transformations.

To infer the type of a method, we find its abstract transformation from T, feed it with a typing for its arguments, and then get the type of the return variable from the resulting typing environment. If we add a new method into the program, we can use T to analyze it as long as the old methods do not depend on it. In this way, we can reuse the existing analysis results to analyze new code rather than reanalyzing the whole program and library.

We note that our approach to type inference via abstract transformations is independent from region types. What essential is the finite lattice structure of types. For example, it also works for class types of the standard Java type system.

Lastly, we have a prototype implementation of the above type inference algorithm for the type system of [ESX21] based on the Soot framework [SG]. It takes a Java bytecode program and a programming guideline as inputs, infers the type and effect of the program, and then verifies if the program adheres to the guideline.

References

[BGH13] Lennart Beringer, Robert Grabowski, and Martin Hofmann. Verifying pointer and string analyses with region type systems. *Computer Languages, Systems & Structures*, 39(2):49–65, 2013.

[EHZ17] Serdar Erbatur, Martin Hofmann, and Eugen Zălinescu. Enforcing programming guidelines with region types and effects. In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems (APLAS 2017)*, volume 10695 of *Lecture Notes in Computer Science*, pages 85–104. Springer, Cham, 2017.

[ESX21] Serdar Erbatur, Ulrich Schöpp, and Chuangjie Xu. Type-based enforcement of infinitary trace properties for Java. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021)*, pages 18:1–18:14. Association for Computing Machinery, 2021.

[GHL12] Robert Grabowski, Martin Hofmann, and Keqin Li. Type-based enforcement of secure programming guidelines — code injection prevention at SAP. In G. Barthe, A. Datta, and S. Etalle, editors, *Formal Aspects of Security and Trust (FAST 2011)*, volume 7140 of *Lecture Notes in Computer Science*, pages 182–197. Springer, Berlin, Heidelberg, 2012.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[KSK07] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems*, 30(1):1–41, 2007.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. *SIGPLAN Notices*, 26(11):146–161, 1991.

[SG] McGill University Sable Group. Soot - A framework for analyzing and transforming Java and Android applications.

Type Theories with Universe Level Judgments

Marc Bezem[1], Thierry Coquand[3], Peter Dybjer[3], and Martín Escardó[2]

[1] University of Bergen, Norway
[2] University of Birmingham, United Kingdom
[3] Chalmers University of Technology, Gothenburg, Sweden

History and state of the art. The system of simple type theory, as introduced by Church [2], is elegant and forms the basis of several proof assistants. However, it has some unnatural limitations: it is not possible in this system to talk about an arbitrary type, or about an arbitrary structure. It is also not possible to form the collection of e.g. all groups, as needed in category theory. In order to address these limitations, Martin-Löf [10, 9] introduced a system with a type V of all types. A function $A \to V$ in this system can then be seen as a family of types over a given type A, and it is natural in such a system to refine the operations of simple type theory, exponential and cartesian product, to operations of dependent products and sums. After the discovery of Girard's paradox in [5], Martin-Löf [11] introduced a distinction between *small* and *large* types, similar to the distinction introduced in category theory between large and small sets, and the type V became the (large) type of small types. The name "universe" for such a type was chosen in analogy with the notion of universes introduced by Grothendieck to represent category theory in set theory.

Later, Martin-Löf [12] introduced a countable tower of universes $\mathsf{U}_0 : \mathsf{U}_1 : \mathsf{U}_2 : \cdots$. We refer to the indices $0, 1, 2, \ldots$ as *universe levels*.

Before the advent of univalent foundations, most type theorists expected only the first few universe levels to be relevant in practical formalisations. This included the expectation that it might be feasible for a user of type theory to explicitly assign universe levels to their types, simply adding updated versions of earlier definitions when they were needed at different levels. However, the number of copies of definitions does not only grow with the level, but also with the number of type arguments in the definition of a type former. (The latter growth can be exponential!)

To deal with this problem Huet [8] and Harper and Pollack [6] and, in Coq, Sozeau and Tabareau [14] introduced *universe polymorphism*. Their "implicit" approach to universe polymorphisms is, however, problematic w.r.t. modularity, as pointed out in [3, 13]: one can prove $A \to B$ in one file, and $B \to C$ in one other file, while $A \to C$ is not valid. In order to cope with this issue, J. Courant [3] suggested to have explicit level universes, with a sup operation (see also [7]). This approach is now followed in Agda and in Voevodsky's proposal [16].

With the advent of Voevodsky's univalent foundations, the need for universe polymorphism has only increased, see for example [16]. The *univalence axiom* states that for any two types X, Y the canonical map

$$\mathsf{idtoeq}_{X,Y} : (X = Y) \to (X \simeq Y)$$

is an equivalence. Formally, the univalence axiom is an axiom scheme which is added to Martin-Löf type theory. If we work in Martin-Löf type theory with a countable tower of universes, each type is a member of some universe U_n. Such a universe U_n is *univalent* provided for all $X, Y : \mathsf{U}_n$ the canonical map $\mathsf{idtoeq}_{X,Y}$ is an equivalence. Let UA_n be the type expressing the univalence of U_n, and let $\mathsf{ua}_n : \mathsf{UA}_n$ for $n = 0, 1, \ldots$ be a sequence of constants postulating the respective instances of the univalence axiom. We note that $X = Y : \mathsf{U}_{n+1}$ and $X \simeq Y : \mathsf{U}_n$ and hence UA_n is in U_{n+1}. If we have a type of levels, as in Agda [15] or Lean [4], we can express universe polymorphism as quantification over universe levels.

We remark that universes are more important in a predicative framework than in an impredicative one. Consider for example the formalisation of real numbers as Dedekind cuts, or domain elements as filters of formal neighbourhoods, which belong to U_1 since they are properties of elements in U_0. However, even in a system using an impredicative universe of propositions, such as the ones in [8, 4], there is a need for the use of definitions parametric in universe levels.

Our contribution. The goal of this work is to complement the proposals by Courant [3] and Voevodsky [16] by handling constraints on universe levels and having instantiation operations. We start by giving the rules for a basic version of dependent type theory with Π, Σ, N, and an identity type former Id. We then explain how to add an externally indexed countable sequence of universes $\mathsf{U}_n, \mathsf{T}_n$ à la Tarski with or without cumulativity rules.

We introduce then an internal notion of universe level and add two new judgment forms: $l\ \mathsf{Level}$ meaning that l is a universe level, and $l = m$ meaning that l and m are equal universe levels. Here level expressions are built up from level variables α using a successor operation l^+ and a join (supremum, maximum) operation $l \vee m$. We let judgments depend not only on a context of ordinary variables, but also on a list of level variables $\alpha_1, \ldots, \alpha_k$, giving rise to a theory with level polymorphism. Certain typing rules are conditional on judgments of the form $l = m$. This is a kind of ML-polymorphism since we only quantify over global level variables.

We then extend the above type theory with formation rules for level-indexed product types $[\alpha]A$ meaning "A is a type for all universe levels α". Furthermore, introduction and elimination rules for such types are given, as well as some new computational rules. An example that uses level-indexed products is the following type which expresses the theorem that univalence for universes of arbitrary level implies function extensionality for functions between universes of arbitrary levels.

$$([\alpha]\mathsf{IsUnivalent}\,\mathsf{U}_\alpha) \to [\beta][\gamma]\mathsf{FunExt}\,\mathsf{U}_\beta\,\mathsf{U}_\gamma$$

We also present (a variation of) Voevodsky's proposal [16] with level constraints, complementing his proposal with a way to instantiate universe polymorphic constants introduced with level variables and constraints. We shortly discuss the decision problems for sup-semilattices with successor that come with this approach. These problems can be solved in polynomial time, as shown in [1].

As an example, we can define in our system a constant

$$c \ :\equiv\ \langle\alpha\ \beta\rangle\lambda_{Y:\mathsf{U}_\beta}\mathsf{Id}\ \mathsf{U}_\beta\ Y\ (\Sigma_{X:\mathsf{U}_\alpha}X \to Y) \quad : \quad [\alpha\ \beta][\alpha < \beta]\mathsf{U}_\beta \to \mathsf{U}_{\beta^+},$$

since $\Sigma_{X:\mathsf{U}_\alpha}X \to Y$ has type U_β in the context

$$\alpha : \mathsf{Level},\ \beta : \mathsf{Level},\ \alpha < \beta,\ Y : \mathsf{U}_\beta.$$

We can instantiate this constant c with two levels l and m, and this will be of type

$$[l < m]\mathsf{U}_m \to \mathsf{U}_{m^+},$$

which only can be used if $l < m$ holds in the current context.

In the current system of Agda [15], the constraint $\alpha < \beta$ is represented indirectly by writing β in the form $\gamma \vee \alpha^+$ and c is defined as

$$c \ :\equiv\ \langle\alpha\ \gamma\rangle\lambda_{Y:\mathsf{U}_{\alpha^+\vee\gamma}}\mathsf{Id}\ \mathsf{U}_{\alpha^+\vee\gamma}\ Y\ (\Sigma_{X:\mathsf{U}_\alpha}X \to Y) \quad : \quad [\alpha\ \gamma]\mathsf{U}_{\alpha^+\vee\gamma} \to \mathsf{U}_{\alpha^{++}\vee\gamma^+},$$

which arguably is less readable. Moreover, not all constraints that occur in practice can be expressed in this way.

References

[1] Marc Bezem and Thierry Coquand. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *TCS*, 913:1–7, 2022.

[2] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] Judicaël Courant. Explicit universes for the calculus of constructions. In Victor Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, volume 2410 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2002.

[4] 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)*, 2015.

[5] Jean-Yves Girard. Thèse d'état. 1971.

[6] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[7] Hugo Herbelin. Type inference with algebraic universes in the Calculus of Inductive Constructions, 2005. unpublished note.

[8] Gérard Huet. Extending the calculus of constructions with Type:Type. unpublished manuscript, April 1987.

[9] Per Martin-Löf. On the strenght of intuitionistic reasoning. Preprint, Stockholm University, 1971.

[10] Per Martin-Löf. A theory of types. Preprint, Stockholm University, 1971.

[11] Per Martin-Löf. An intuitionistic theory of types. Preprint, Stockholm University, 1972.

[12] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North Holland, 1975.

[13] Carlos Simpson. Computer theorem proving in mathematics. *Letters in Mathematical Physics*, 69(1-3):287–315, Jul 2004.

[14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In *Interactive Theorem Proving - 5th International Conference (ITP 2014)*, 2014.

[15] Agda team. The Agda manual. `https://agda.readthedocs.io/en/v2.6.2.1/`.

[16] Vladimir Voevodsky. Universe polymorphic type system. http://www.math.ias.edu/Voevodsky/voevodsky-publications_abstracts.html#UPTS, 2014.

TypOS: An Operating System for Typechecking Actors

Guillaume Allais[1], Malin Altenmüller[2], Conor McBride[2], Georgi Nakov[2], Fredrik Nordvall Forsberg[2], and Craig Roy[3]

[1] University of St Andrews, Fife, United Kingdom
[2] University of Strathclyde, Glasgow, United Kingdom
[3] Quantinuum & Cambridge Quantum, Cambridge, United Kingdom

Introduction We report work in progress on TypOS, a domain-specific language for experimenting with typecheckers and elaborators. Our remit is similar to those of other domain-specific languages such as Andromeda [1], PLT Redex [3], or Turnstyle+ [2]. However, we try to minimise demands on the order in which subproblems are encountered and constraints are solved: when programs contain holes, and constraints are complex, it helps to be flexible about where to make progress. TypOS tackles typing tasks by spawning a network of concurrent actors [4], each responsible for one node of the syntax tree, communicating with its parent and child nodes via channels. Constraints demanded by one actor may generate enough information (by solving metavariables) for other actors to unblock. Metavariables thus provide a secondary and asynchronous means of communication, creating subtle difficulties modelling actor resumptions as host-language functions, because metavariables unsolved at the time of a resumption's creation may be solved by the time it is invoked. Thus forced into a more syntactic representation of suspended processes, we decided to explore what opportunities a new language could bring.

Term syntax and judgement forms TypOS supports a generic LISP-style syntax for terms, including atoms ('a), cons lists (`[`a b c`]`), and an α-invariant notion of term variables (x) and binding ($\backslash x.t$). Users can restrict the shape of such terms using context-free grammars, for example defining the syntactic categories of simple types (`'Type`), synthesisable (`'Synth`), and checkable terms (`'Check`). The notion of *judgement form* is recast as channel *interaction protocol* [5], specifying what to communicate, and in which direction. We declare an actor by giving its name and protocol. For example, we may declare type checking and synthesis actors:

```
check : ?'Type. ?'Check. -- receives a type, then a checkable term
synth : ?'Synth. !'Type. -- receives a synthesisable term, sends a type back
```

Information about free variables is kept in contexts, which must also be declared, for example:

```
ctxt |- 'Synth -> 'Type  -- declare ctxt to map 'Synth variables to types
```

Typing rules For each judgement form, we define an actor as the server for all the rules whose conclusion takes that form. Where rules have premises, a typing actor spawns children, each with its own channel. Actors may fork parallel subactors, generate fresh metavariables, declare constraints, bind local variables, match on terms using a scope-aware pattern language, and extend and query contexts. For example, here are actors for bidirectional type checking and type synthesis of the simply typed lambda calculus, following the interaction protocol above:

```
check@p = p?ty. p?tm. case tm -- receive type and term via p, match on tm
  { ['Lam \x. body] ->        -- lambda case:
     'Type?S. 'Type?T.        --   make fresh type metavars S and T
     ( ty ~ ['Arr S T]        --   in parallel: constrain ty as arrow type
```

```
    | \x.                    --   and bring fresh x into scope
      ctxt { x -> S }.       --   then extend ctxt to map x to S
      check@q. q!T. q!body.) --   then spawn child on channel q to check body
  ; ['Emb e] ->              -- embedded synthesisable term e:
    synth@q. q!e. q?S.       --   spawn child to synthesise type S for e
    S ~ ty }                 --   constrain S as ty

synth@p = p?tm. if tm in ctxt -- receive term tm via p, query ctxt for tm
  { S -> p!S. }               -- if tm is a variable in ctxt, send its type
  else case tm                -- otherwise match on tm
  { ['Ann t T] ->             -- type annotated term case: in parallel
    ( type@q. q!T.            --   spawn child to validate type T
    | check@r. r!T. r!t.      --   spawn child to check t has type T
    | p!T. )                  --   send type T
  ; ['App f s] ->             -- application case:
    'Type?S. 'Type?T.         --   make fresh type metavars S and T
    ( synth@q. q!f. q?F.      --   spawn child to synthesise a type F for f
      F ~ ['Arr S T]          --   constrain F as arrow type
    | check@r. r!S. r!s.      --   spawn child to check argument
    | p!T. ) }                --   send the target type back
```

Schematic variables and scope management The notion of *schematic variable* in a typing rule is recast as the ordinary notion of program variable (e.g. `ty`, `tm`, `S`, `T` above) within an actor. We are careful to distinguish these 'actor variables' from both 'term variables' of the syntax being manipulated (e.g. `x`), and 'channel variables' (e.g. `p`, `q`, `r`). Schematic variables in typing rules are usually thought of as implicitly universally quantified: by contrast, each of our actor variables has one explicit binding site in an actor process, either at an input action, or in a pattern-match. As in Delphin [8], the term variables in scope at each point in an actor are determined by explicit binding constructs. The scope for the signals on a channel is bounded by the scope at its creation, ensuring that parents never encounter variables bound locally by their children. Internally, we use a precisely scoped co-deBruijn representation of terms [7].

Metavariables and constraints The actor model principle of sharing by message not memory lets us guarantee that each actor has direct knowledge of only those term variables it has bound itself. Actors cannot learn the names of any term variables free when they were spawned, and are thus less likely to violate stability under substitution. Nonetheless, we do support the one form of memory that distributed concurrent processes may safely share [6]: the metavariables actors create and share mutate monotonically only by becoming more defined. Actors cannot detect that a metavariable is unsolved, but block when matching strictly on one. Decisions already taken on the basis of less information need never be retracted when more arrives.

Executing actors In the future, we will implement a concurrent runtime, but for the moment, we use a stack-based virtual machine. Each actor runs until it blocks, then the machine refocuses on the next place progress can be made, until execution stabilises. The derivation thus constructed can readily be extracted from the final configuration of the stack.

Try it yourself TypOS is available at https://github.com/msp-strath/TypOS, together with more examples of actors.

References

[1] Andrej Bauer, Philipp G. Haselwarter, and Anja Petkovic. Equality checking for general type theories in andromeda 2. In Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff, editors, *ICMS '20*, volume 12097 of *Lecture Notes in Computer Science*, pages 253–259. Springer, 2020.

[2] Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. Dependent type systems as macros. *Proc. ACM Program. Lang.*, 4(POPL):3:1–3:29, 2020.

[3] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009.

[4] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI '73*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[5] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93*, pages 509–523. Springer, 1993.

[6] Lindsey Kuper. *Lattice-Based Data Structures For Deterministic Parallel And Distributed Programming.* PhD thesis, Indiana University, 2015.

[7] Conor McBride. Everybody's got to be somewhere. In Robert Atkey and Sam Lindley, editors, *MSFP '18*, volume 275, pages 53–69, 2018.

[8] Adam Poswolsky and Carsten Schürmann. System description: Delphin – A functional programming language for deductive systems. *Electron. Notes Theor. Comput. Sci.*, 228:113–120, 2009.

Unifying Cubical and Multimodal Type Theory

Frederik Lerbjerg Aagaard[1], Magnus Baunsgarrd Kristensen[2], Daniel Gratzer[1], and Lars Birkedal[1]

[1] Aarhus University, Denmark
[2] IT University of Copenhagen, Denmark

We introduce cubical multimodal type theory ($\mathsf{MTT}_\square$), a dependent type theory that combines multimodal type theory (MTT) [GKNB20] with cubical type theory (CTT) [CCHM18]. The result not only retains the desirable qualities of both theories but also validates an extensionality principle for modal types that is not present in MTT. For semantics, we provide an axiomatic approach to constructing models, including presheaf models. As an example, we model guarded recursion by replaying the arguments from [GKNB20] now in $\mathsf{MTT}_\square$. Using presheaf models, we prove that Löb induction is consistent with the theory, and we prove using modal extensionality that Löb induction gives a propositionally unique fix-point which in MTT requires additional axioms [GKNB21].

Cubical type theory. CTT [CCHM18] was introduced to achieve two things: a computationally effective interpretation of the univalence axiom and a more well-behaved identity type satisfying e.g. function extensionality. It extends MLTT with an interval object $\mathbb{I}$—an abstraction of the interval $[0,1]$—and *path types* $\mathsf{Path}_A(a_0,a_1)$, essentially the type of functions from the interval that agree on endpoints. A path corresponds to a term that depends on a single interval variable but dependence on multiple variables yields squares, cubes, or n-cubes. The intent is for path types to replace identity types, but they are not yet transitive. To fix this, CTT includes *Kan operations* which state that if a path is defined on only a part of an n-cube and one endpoint can be extended to the whole cube then the other endpoint may be extended as well. With this, one can prove that paths can be composed, resulting in transitivity and support for path induction (though with computation only up to a path). In order for the Kan operations to compute, they must have computation rules for every type, e.g. there is a rule specifying how the Kan operation in $A \times B$ can be reduced to operations in A and B.

Multimodal type theory. Separately, it is common to increase the expressivity of MLTT by adding *modalities* [BMSS12], but proving that these extensions satisfy desirable qualities like normalisation is laborious. MTT [GKNB20] alleviates this problem by providing a single type theory that is parametrised by a *mode theory*—a 2-category that specifies the modal situation—yet satisfies canonicity [GKNB20] and normalisation [Gra22]. By instantiating MTT with an appropriate mode theory, we can model specific modal type theories, e.g. guarded recursion [GKNB20]. Each object of a mode theory $(m,n,\ldots)$, called a mode, is a copy of MLTT, whilst each 1-cell $(\mu,\nu,\ldots)$, called a modality, allows movement between modes. Thus, for any modality $\mu : n \longrightarrow m$, we have amongst others the rules:[1]

$$\frac{\Gamma\ \mathsf{cx} \mathbin{@} m}{\Gamma, \{\mu\}\ \mathsf{cx} \mathbin{@} n} \qquad \frac{\Gamma, \{\mu\} \vdash A \mathbin{@} n}{\Gamma \vdash \langle \mu \mid A \rangle \mathbin{@} m} \qquad \frac{\Gamma, \{\mu\} \vdash a : A \mathbin{@} n}{\Gamma \vdash \mathsf{mod}_\mu(a) : \langle \mu \mid A \rangle \mathbin{@} m}$$

[1] The left-most rule is referred to as *locking* a context.

Cubical multimodal type theory. Cubical multimodal type theory ($\mathsf{MTT}_\square$) is a combination of MTT and CTT. Like MTT it is parametrised by a mode theory, but whereas MTT contains a copy of MLTT at each mode, $\mathsf{MTT}_\square$ contains a copy of CTT. Each instance of $\mathsf{MTT}_\square$ thus consists of a number of copies of CTT connected by weak dependent right adjoints.

The challenge in this combination is that in order for terms to compute, computation rules for interactions between modal and cubical aspects must be added; in particular, a computation rule for Kan operations in modal types. However, this rule will not be well-typed before adding *exchange principles*, governing interactions between the cubical and modal aspects of the theory. The exact makeup of these principles is a subtle part of the design of this theory, and care has to be taken to encapsulate the desired examples.

We adopt a principle of orthogonality, i.e. that modal and cubical aspects should interfere minimally with each other. Concretely, we have rules stating that a dimension term $\Gamma \vdash r : \mathbb{I}_m \mathbin{@} m$ may be moved to a locked context $\Gamma, \{\mu\} \vdash r^\mu : \mathbb{I}_n \mathbin{@} n$ and the same for faces. This induces substitutions $\Gamma, i : \mathbb{I}_m, \{\mu\} \vdash \sigma_\mu : \Gamma, \{\mu\}, i : \mathbb{I}_n \mathbin{@} n$ and $\Gamma, \phi, \{\mu\} \vdash \tau_\mu : \Gamma, \{\mu\}, \phi^\mu \mathbin{@} n$, where $-, \phi$ is restriction of a context to the face ϕ, which we demand are isomorphisms. A similar approach to combining CTT with a modal type theory is taken in [KMV21], whilst [Cav21] and [MV18] use equalities instead of isomorphisms.

Using these operations, we establish a computation rule for Kan operations in $\langle \mu \mid A \rangle$ in terms of A, which we prove to be well-typed; concretely:

$$\mathsf{comp}^i_{\langle \mu \mid A \rangle} \left[\phi \mapsto \mathsf{mod}_\mu(u)\right] \mathsf{mod}_\mu(u_0) = \mathsf{mod}_\mu(\mathsf{comp}^i_A \left[\phi^\mu \mapsto u[\sigma_\mu \circ \tau_\mu]\right] u_0)$$

Just as CTT validates many extensionality principles, with these exchange principles, we get for $\mathsf{MTT}_\square$ modal extensionality:

Theorem 1. *Given a modality $\mu : n \longrightarrow m$ and terms $A : \mathsf{U} \mathbin{@} n$ and $a, b : \mathsf{El}(A) \mathbin{@} n$, there is a path equivalence $\langle \mu \mid \mathsf{Path}_{\mathsf{El}(A)}(a, b) \rangle \simeq \mathsf{Path}_{\langle \mu \mid \mathsf{El}(A) \rangle}(\mathsf{mod}_\mu(a), \mathsf{mod}_\mu(b)) \mathbin{@} m$.*

$\mathsf{MTT}_\square$ is formally defined as a *generalised algebraic theory*, and it, therefore, induces a category of models, including an initial model. Due to the complexity of the type theory, constructing such a model is a laborious task, and we, therefore, introduce *cubical MTT cosmoi* as an axiomatic approach to producing models. These assign a topos satisfying axioms from [OP18] and [LOPS18] to each mode while each modality is assigned to an adjunction which induces a dependent right adjoint whose left adjoint preserves the cubical structure coherently. These axioms ensure that each mode models CTT while the entire structure models MTT. Finally, by requiring that the cubical structure is appropriately preserved these models validate the aforementioned exchange principles.

Theorem 2. *Any cubical MTT cosmos induces a model of $\mathsf{MTT}_\square$.*

Theorem 3. *Let $f : \mathcal{M} \longrightarrow \mathbf{Cat}$ be a strict 2-functor, and write $F^*(\mu)$, $F_!(\mu)$, and $F_*(\mu)$ for the precomposition, left Kan extension, and right Kan extension respectively of $f(\mu) \times \mathsf{id}_\square$. Then:*

- *the network of morphisms of LOPS topoi given by the adjunctions $F^*(\mu) \dashv F_*(\mu)$ induces a model of $\mathsf{MTT}_\square$ over $\mathcal{M}$; and*
- *the network of morphisms of LOPS topoi given by the adjunctions $F_!(\mu) \dashv F^*(\mu)$ induces a model of $\mathsf{MTT}_\square$ over $\mathcal{M}$.*

Acknowledgements. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

References

[BMSS12] Lars Birkedal, Rasmus Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:1)2012.

[Cav21] Evan Cavallo. *Higher inductive types and internal parametricity for cubical type theory*. PhD thesis, Carnegie Mellon University, 2021.

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8475, doi:10.4230/LIPIcs.TYPES.2015.5.

[GKNB20] Daniel Gratzer, G.A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20. ACM, 2020. doi:10.1145/3373718.3394736.

[GKNB21] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal Dependent Type Theory. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021. URL: https://lmcs.episciences.org/7713, doi:10.46298/lmcs-17(3:11)2021.

[Gra22] Daniel Gratzer. Normalization for multimodal type theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3531130.3532398.

[KMV21] Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. Greatest hits: Higher inductive types in coinductive definitions via induction under clocks. 2021. URL: https://arxiv.org/abs/2102.01969, arXiv:2102.01969.

[LOPS18] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal Universes in Models of Homotopy Type Theory. In H. Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 22:1–22:17. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. arXiv:1801.07664, doi:10.4230/LIPIcs.FSCD.2018.22.

[MV18] Rasmus Ejlers Møgelberg and Niccolò Veltri. Bisimulation as path type for guarded recursive types. *CoRR*, abs/1810.13261, 2018. URL: http://arxiv.org/abs/1810.13261, arXiv:1810.13261.

[OP18] Ian Orton and Andrew M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. *Logical Methods in Computer Science*, 14(4), 2018. arXiv:1712.04864, doi:10.23638/LMCS-14(4:23)2018.

Validating OCaml soundness by translation into Coq

Jacques Garrigue and Takafumi Saikawa

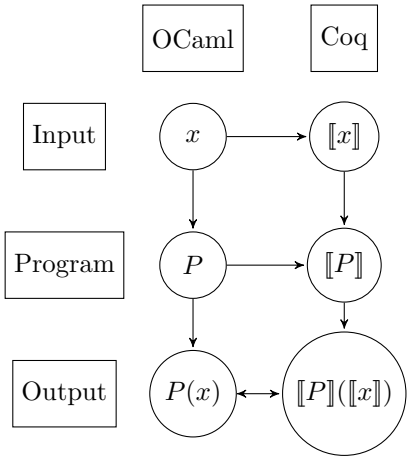
Graduate School of Mathematics, Nagoya University

The programming language OCaml is known for the expressiveness of its type system. Starting with an ML core consisting of algebraic datatypes, mutable references, and exceptions, it extends it with polymorphic variants and objects, extensible datatypes, GADTs, a module system, and even first class modules. Further extensions are planned.

Proving the correctness of both theory and implementation for such a complex system is a daunting task. Formalization of the theory of ML languages started in the 1990's, both in HOL [13], and Coq [1], and culminated with the formalization of full SML in Twelf [7]. However, this is only with Cake ML that the proof included the actual implementation [6, 12], albeit for a much smaller language. Concerning OCaml, most parts of the type system were proved correct on paper, but only independently of each other. Some parts were further formalized in Coq [4, 8], but independently of the actual implementation, which often differs in subtle ways. A formal semantics for the core of the language has also been given, but again independently of the implementation [9].

An alternative to actually proving things is to check them a posteriori. That is, rather than proving the OCaml typechecker correct, verify that its output is correct. That is the approach taken by Couderc [3], by writing a type checker to be applied to the so called *typed tree*, which is the result of type inference. While redundancy provides some extra guarantees against simple implementation bugs, this approach may still leave some bugs, and does not ensure that the type system itself is sound.

We propose here a new approach, based on the translation of OCaml programs to Gallina. By implementing it as an extra backend to the OCaml compiler, taking the *typed tree* as input, we can use Coq's type checker to ensure the type soundness of source programs, as long as we can assume that the semantics is unchanged. The type soundness is a consequence of Coq's subject reduction property, following the principle shown in the following figure.



Namely, if for all $P : \tau \to \tau'$ and $x : \tau$ we have:

1. P translates to $[\![P]\!]$, and $\vdash [\![P]\!] : [\![\tau \to \tau']\!]$
2. x translates to $[\![x]\!]$, and $\vdash [\![x]\!] : [\![\tau]\!]$
3. $[\![P]\!]$ applied to $[\![x]\!]$ evaluates to $[\![P(x)]\!]$
4. $[\![\cdot]\!]$ is injective on types

then the soundness of Coq's type system implies the soundness of OCaml's evaluation (i.e., evaluation of well-typed programs cannot go wrong).

This approach has a relatively large trusted base: we assume that the translator does not change the semantics of the input program, when using Coq normalization, and that Coq indeed enjoys type soundness [10].

The goal of relying on Coq for the soundness has led us to a number of design choices. Since we want to be able to evaluate programs inside Coq, we realize side effects through a concrete monadic implementation, and avoid using axioms that would block evaluation. We

also avoid unsound extensions of Coq as much as possible. Finally, as references (and polymorphic comparison) require some form of dynamic type information, we provide an intensional representation of types.

Extraction from Coq to other languages has been an active research area since its beginning, but the opposite direction is much more recent. Currently, two other translators are available: `coq-of-ocaml` [2] and `hs-to-coq` [11], for OCaml and Haskell respectively. However, their goal is different from ours, as they intend to prove properties on the translated programs, and as a result are happy to restrict themselves to a subset of the language in order to get simpler translated code. In particular they do not introduce an intensional representation of types.

The current version of the translator [5] only supports the core part of the language: Hindley-Milner polymorphism, algebraic datatypes, polymorphic comparison, references and exceptions. However, we already have extensions in mind, and in particular the choice of having an intensional representation of types should also be an advantage when translating GADTs, as they rely on the ability to compare types, which in Coq is only possible if they have a matchable representation. Due to our need to aggregate all defined types, the translator is restricted to single-file programs, but this restriction should also be eventually lifted, by adding a linking phase.

Since we intend to translate faithfully arbitrary OCaml programs to Coq, we need to recreate the OCaml world inside Coq. The two essential parts of it are:

- defining a monad $M\ T = Env \to Env \times (T + Exn)$ for state (references), exceptions, and non-termination (seen as a non-catchable exception),
- defining a translation from the intensional representation of types to their Coq semantics.

Here we will just show the essential part of the functor `REFmonad` that creates such a monad from the intensional representation of types `ml_type` and its translation `coq_type`.

```
Module Type MLTY.
  Parameter ml_type : Set.
  Parameter ml_exn : ml_type.
  Parameter coq_type : forall M : Type -> Type, ml_type -> Type.
End MLTY.
Module REFmonad(MLtypes : MLTY).
  Record key := mkkey {key_id : int; key_type : ml_type}.
  Record binding (M : Type -> Type) := mkbind
    { bind_key : key; bind_val : coq_type M (key_type bind_key) }.
  Definition M0 Env Exn T := Env -> Env * (T + Exn).
  #[bypass_check(positivity)]                (* non-positive definition *)
  Inductive Env := mkEnv : int -> seq (binding (M0 Env Exn)) -> Env.
       with Exn := Catchable of coq_type (M0 Env Exn) ml_exn
                 | GasExhausted | RefLookup | BoundedNat.
  Definition M T := Env -> Env * (T + Exn).          (* = M0 Env Exn T *)
  ... (* Monadic operations for references and exceptions *)
End REFmonad.
```

You can see that `coq_type` itself depends on the monad, as it is needed when translating function types. As a result, the mutually recursive definition of `Env` and `Exn` clearly violates the positivity condition enforced by Coq. This comes as no surprise, since it is well known that references can be used in ML to define non-terminating functions, in the absence of recursion. However, we conjecture that disabling the positivity check for this definition alone does not endanger Coq soundness, as non-termination is restricted to computations happening inside the monad; namely, we can indeed create values of type `M False`, but we cannot extract `False` from the resulting sum type.

References

[1] Olivier Boite and Catherine Dubois. Proving type soundness of a simply typed ML-like language with references. In *Proc. of the International Conference on Theorem Proving in Higher Order Logics*, 2001.

[2] Guillaume Claret. Coq of OCaml. In *OCaml Users and Developers Meeting*, August 2014.

[3] Pierrick Couderc. *Vérification des résultats de l'inférence de types du langage OCaml*. PhD thesis, Université Paris-Saclay, October 2018.

[4] Jacques Garrigue. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, 25(4):867–891, November 2014.

[5] Jacques Garrigue. OCaml in Coq. GitHub PR, 2022. https://github.com/COCTI/ocaml/pull/3.

[6] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proc. 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 173–184, New York, NY, USA, 2007. Association for Computing Machinery.

[8] Thomas Leventis. A row Coq proof of soundness for the relaxed value restriction. https://www.irif.fr/~leventis/pub/Stages/Coq/, September 2012.

[9] Scott Owens. A sound semantics for OCaml light. In *Proc. European Symposium on Programming*, volume 4960 of *Springer LNCS*, pages 1–15, April 2008.

[10] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), January 2020.

[11] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. In *Proc. International Conference on Functional Programming*, pages 14—-27, New York, NY, USA, 2018.

[12] Yong Kiam Tan, Scott Owens, and Ramana Kumar. A verified type system for CakeML. In *Proc. 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Myra Ellen VanInwegen. *The machine-assisted proof of programming language properties*. PhD thesis, University of Pennsylvania, August 1996.

Verification Techniques for Smart Contracts in Agda

Fahad F. Alhabardi[1], Arnold Beckmann[2], Bogdan Lazar[3], and Anton Setzer[4]

[1] Swansea University, Dept. of Computer Science
fahadalhabardi@gmail.com
[2] Swansea University, Dept. of Computer Science
a.beckmann@swansea.ac.uk
https://www.beckmann.pro/
[3] University of Bath
lazarbogdan90@yahoo.com
[4] Swansea University, Dept. of Computer Science
a.g.setzer@swansea.ac.uk
http://www.cs.swan.ac.uk/~csetzer/

Abstract

We have shown in [1] how some of the standard Bitcoin scripts can be verified using Hoare Logic in the interactive theorem prover Agda. Our framework was developed for standard instructions and in addition the multisig and the time delay instructions. We developed two ways of establishing human-readable weakest preconditions: (1) A step-by-step approach of working backwards in the program and (2) symbolic execution of the program and determining the accepting paths.

In this presentation, we investigate how these two approaches can be extended to Bitcoin scripts that use non-local instructions such as `OP_IF`, `OP_ELSE`, and `OP_ENDIF`. Our approach is based on a basic operational semantics [12], which added an additional stack called lfStack to the standard stack.

One of the most notable trustless and decentralised architectures is given by blockchain technology. It runs in a trustless environment using immutable peer-to-peer technology combined with a consensus protocol. Cryptocurrencies, which may be combined with smart contracts, are based on blockchain technology. Blockchain technology enables data to be shared universally in a secure, trusted, and synchronised way [15].

Satoshi Nakamoto [11] launched Bitcoin in 2008 as the first cryptocurrency. Bitcoin enables private, anonymous payments over a peer-to-peer network [9]. Since then, several cryptocurrencies have been introduced such as Ethereum [6]. This resulted in a new and emerging era of decentralised and digital monetary systems in which there is no need for any central entity such as central banks to manage and control users' transactions.

Smart contracts are another intriguing topic that has emerged as a result of the new era of blockchain [13, 10]. Smart contracts are defined as programs that are performed automatically when specific circumstances are met. Because smart contracts are capable of locking high monetary values, they are business-critical systems, hence techniques for evaluating their security and validity are required [10, 14]. In Bitcoin, smart contracts are written in the language Script. [5]. Other cryptocurrencies have their own languages [2]. One should note that the Ethereum Virtual Machine (EVM), into which smart contracts of Ethereum's smart contract languages are compiled, is as Bitcoin Script based on a stack machine[1], so we expect that techniques used for verifying Bitcoin scripts carry over when verifying smart contracts of the EVM.[2]

[1] [3, Ch 13]: "The EVM has a stack-based architecture, storing all in-memory values on a stack."

[2] There are as well many differences: The EVM is Turing complete and has jumps, has access to the state, and can make calls to other contracts. See the following blogposts/forum discussions comparing the two machines: [8, 7]

Hoare triples have been used to validate the correctness and capability of smart contract programs. In our approach we will verify these Hoare triples using Agda as our proof assistant for two reasons: (1) Agda is both a programming language and a theorem prover, which means we can both execute and verify smart contracts in Agda. (2) Proofs in Agda can be checked by hand. Smart contracts must be carefully verified because of high monetary values associated with them. It might not be sufficient to provide trust in the correctness of a smart contract to have a proof in a theorem prover, having to rely on its correct implementation – attackers might use an error in the theorem prover in order to create a false proof of the correctness of a compromised smart contract. If the amount of money being protected by a Bitcoin script is high, in addition to machine checking manual checking of correctness proofs might be appropriate.

In our previous article [1] we defined and formalised an operational semantics of a fragment of Bitcoin Script using instructions having only local behaviour. This included basic instructions and some more complex ones: the multi-signature instruction and an instruction enforcing a time delay. Then we verified those scripts using weakest preconditions for Hoare triples. The operational semantics was based on a state $\mathsf{StackState} := \mathsf{Time} \times \mathsf{Msg} \times \mathsf{Stack}$ (more precisely in Agda an explicit record type was used) formed from the normal stack Stack, a message Msg representing the message of the transaction to be signed, and the current time Time in Agda.

In this talk we will investigate the addition of conditional instructions to our fragment of Bitcoin Script, which have a non-local behaviour, and will discuss how to verify scripts written in this extended language. As discussed in [12], the use of control follow statements can be dealt with in the operational semantics by using a second stack IfStack, which keeps track of the nesting of conditionals. The new state is $\mathsf{Time} \times \mathsf{Msg} \times \mathsf{Stack} \times \mathsf{IfStack}$ defined as a record type State in Agda. Our approach is different from the usual approach of converting programs in Forth involving conditionals into programs with jumps. Instead, we work directly with the unstructured machine programs, and we adopted the usual techniques in Hoare logic for dealing with conditionals to such unparsed programs.

Conditionals can result in an exponential increase of the number of paths in a program. This is a well-known problem in Bitcoin: For instance, Antonopoulos [4, Ch. 7] writes: "Bitcoin Script flow control can be used to construct very complex scripts with hundreds or even thousands of possible execution paths. There is no limit to nesting, but consensus rules impose a limit on the maximum size, in bytes, of a script".

In [1] we argued that the correctness of Bitcoin scripts can be considered as verification of the security of access control systems. Access control is used to restrict access to a resource which in our case is access to the cryptocurrency in question, i.e. bitcoins. Hoare logic, which is based on preconditions and postconditions, is particularly well suited for safety-critical systems with a controllable set of inputs. Additionally, it enables interaction between multiple procedures within a program. When dealing with access control, weakest preconditions are more suitable: The precondition must not only be sufficient but as well necessary to ensure that the post-condition (which expresses access to the resource) is fulfilled after execution of the program.

As in [1] we use two methods for obtaining a human-readable weakest precondition that can be used to assist developers of smart contracts in Bitcoin. The first method is a step-by-step approach that works backwards through the program instruction by instruction. The second method is symbolic execution of the code and conversion to a nested case distinction, which enables reading off the weakest preconditions as the disjunction of the accepting paths. These methods have been formalised in Agda and as an example we apply them to the verification of P2PKH and P2MS, and combinations of them with conditionals. Since we obtain human readable weakest preconditions, these methods allow to reduce the validation gap between user requirements and formal specifications.

References

[1] Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer. Verification of Bitcoin Script in Agda using weakest preconditions for access control, 2022. URL: https://arxiv.org/abs/2203.03054, arXiv:arXiv:2203.03054, doi:10.48550/ARXIV.2203.03054.

[2] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, 67:1–19, 2020. doi:10.1016/j.pmcj.2020.101227.

[3] Andreas Antonopoulos and Gavin Wood. *Mastering Ethereum. Building smart contracts and Dapps*. O'Reilly Media, November 2018.

[4] Andreas M Antonopoulos. *Mastering Bitcoin: Programming the open blockchain*. O'Reilly, 2nd edition, 2017.

[5] Harris Brakmić. Bitcoin Script. In *Bitcoin and Lightning Network on Raspberry Pi: Running Nodes on Pi3, Pi4 and Pi Zero*, pages 201–224, Berkeley, CA, 2019. Apress. doi:10.1007/978-1-4842-5522-3_7.

[6] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. White paper. URL: https://ethereum.org/en/whitepaper.

[7] Differences/similarities of "Bitcoin script" and "Ethereum smart contract"?, Retrieved 9 March 2022. Forum discussion. URL: https://tinyurl.com/2p9xr9cc.

[8] From bitcoin script engine to Ethereum virtual machine, Retrieved 9 March 2022. Blogpost. URL: https://tinyurl.com/28yubhm3.

[9] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in Bitcoin and Ethereum Networks. In *Financial Cryptography and Data Security*, pages 439–457, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg. doi:10.1007/978-3-662-58387-6_24.

[10] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978309.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, 2008. URL: https://www.debr.io/article/21260-bitcoin-a-peer-to-peer-electronic-cash-system.

[12] Anton Setzer and Bogdan Lazar. Modelling smart contracts of Bitcoin in Agda, June 2021. In Henning Basold (Ed): TYPES 2021 – Book of Abstracts, 27th International Conference on Types for Proofs and Programs on 14 - 18 June 2021. URL: https://types21.liacs.nl/download/modelling-smart-contracts-of-bitcoin-in-agda/.

[13] Nick Szabo. Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, 18(2), 1996. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[14] Maximilian Wohrer and Uwe Zdun. Smart Contracts: Security patterns in the Ethereum ecosystem and Solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8, 2018. doi:10.1109/IWBOSE.2018.8327565.

[15] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564, 2017. doi:10.1109/BigDataCongress.2017.85.